FUNCTIONS I (UNDERSTANDING , READING)

ADMIN STUFF

- Reminder that HW2: Personality Quiz has been released
  - Practice due 9/22 (tomorrow!)
  - Project due 9/28
  - Presentations after Fall Break (but sign up by 9/28!)
- It is now time to think about scheduling accomodations through Weingarten for Midterm 1. I cannot do this for you or help you if you forget.
- Midterm 1 exam review in 2 weeks

What are functions?

Functions are named groups of statements/code

- Functions have names, which we can use to call them
- Functions have inputs, which we can use to...
  - control the specific way that they behave on different calls
  - process different information with them on different calls
- Functions can (but don't have to!) return values

```
result = max(17, 34)
```

`max` is the name of a function. We can call it like `max(17, 34)`, which passes in `17` and `34` as its two inputs. The value returned by this function call is `34`, which is saved in `result`

FUNCTION CALLS & RETURN VALUES

An **expression**, as we know, is a bit of Python that has a value.

- `3` is the expression with the value of `3`
- `3 + 7` is the expression with the value of `10`
- `max(7, 11)` is an expression with a value of `11`

When you write a function **call** as part of an expression, that function call gets evaluated in order to determine the full value of the expression.

- Function calls are evaluated inside-out: evaluate the inputs, then call the function with those evaluated inputs.
- This also applies to nested function calls.

Remember this from last class? (Write the values printed by these snippets)

```
# 1
print(len("hello"))
#2
print(max(3 + 4, 10 - 2))
#3
message = "pYTHon rUles".upper()
words = message.split()
print(words)
#4
print("pYTHon rUles".upper().split())
#5
print(min(7, max(len("yessir"), 12 // 5)))
```

SO HOW CAN WE BREAK DOWN WHAT A FUNCTION DOES?

ANATOMY OF A FUNCTION

- **Function definitions** consist of the function's signature as well as a block of statements called its **body**
  - A **function signature** consists of:
    - the function's name
    - the list of parameters that it takes as input.

DISSECTING A FUNCTION

```
def multiply_two_numbers(a, b):
    print(f"Multiplying {a} x {b}!")
    product = a * b
    return product
```

THE SIGNATURE:

```
def multiply_two_numbers(a, b):
```

- `def`, the function's name (`multiply_two_numbers`), a pair of parentheses, and a comma-separated list of parameters (`a` and `b`)
- Use the signature to understand *what is the goal of the function* and *what/how many parameters does the function accept*

THE BODY:

```
    print(f"Multiplying {a} x {b}!")
    product = a * b
    print(product)
```

- multiple statements
- all indented one level relative to signature
- uses `a` and `b` as variables without declaring!
- can end with a `return` statement to produce a value (this example doesn't)
- Use the body to understand *what the function actually does, step by step*

EXAMPLE

A set of three questions **[C1]**:

1. What part of this function is the signature? The body?
2. In one sentence, describe what you think this function does. How do you know?
3. What *data type* do you think this function is meant to take as a parameter? Are there any other types that will work?

```
def print_every_other_char(x):
  for i in range(0, len(x), 2):
    print(x[i])
```

TYPE HINTS

```python
def print_every_other_char(x):
  for i in range(0, len(x), 2):
    print(x[i])
```

👇👇👇👇

```python
def print_every_other_char(x: str):
  for i in range(0, len(x), 2):
    print(x[i])
```

WHAT IF THAT INFO ISN'T ENOUGH?

Sometimes you run into functions that are hard to decipher.

In these instances it helps to walk through the code with some potential input to see what happens.

EXAMPLE:

```
def uhoh(whatami):
  return whatami*whatami
```

HANG ON, WHAT DOES RETURN DO? (BRIEF INTERLUDE)

Function calls are themselves *expressions*, meaning that they always have a value.

- The value of a function call is determined by the value that function **returns**

`return` is keyword that serves two purposes:

- stops function execution in its tracks
- provides a value for the expression of the function call

`return` =/= `print`

EXAMPLE

```
print(max(3 + 4, 10 - 2))
```

Here, `max()` can be defined as the following

```
def max(a, b):
  if b > a:
    return b
  return a
```

So in our first expression, we can replace the value of `max(3 + 4, 10 - 2)` with it's returned value (in this case `8`)

BACK TO THIS: WHAT IF THAT INFO ISN'T ENOUGH?

Sometimes you run into functions that are hard to decipher.

In these instances it helps to walk through the code with some potential input to see what happens.

> **EXAMPLE:**

```
def uhoh(whatami):
  return whatami*whatami
```

PRACTICE READING FUNCTIONS [L1]

Work with a partner to decipher what the following mystery function does:

```
def mystery(x : str):
  i = 0
  new_str = ""
  while i < len(x):
    if x[i] != 'x':
      new_str = new_str + x[i]
    i += 1
  return new_str
```

PRACTICE READING FUNCTIONS [L2]

Work with a partner to decipher what the following mystery function does:

```
def mystery(x : list[int]):
  row = ""
  for i in x:
    for j in x:
      value : int = i * j
      row = row + str(value) + " "
    print(row)
    row = ""
```

Variables defined inside a function only exist within it

This is a concept called **scope**

PRACTICE [C2]

What's wrong with the following code?

```
def factorial(x : int):
  total = 1
  for i in range(x):
    total = total * (i + 1)



x = input("What value would you like to find the factorial of? ")
factorial(int(x))
print(f"{x}! is equal to {total}")
```

ACTIVITY [C3]

How would you fix it?

```
def factorial(x : int):
  total = 1
  for i in range(x):
    total = total * (i + 1)



x = input("What value would you like to find the factorial of? ")
factorial(int(x))
print(f"{x}! is equal to {total}")
```

Choose a better name for each of the four functions below. Each function is run with a single list as its input, e.g. `M1([3, 9, 0, 14])` **[L3]**

```
def M1(lst):
  smallest = lst[0]
  for elem in lst:
    if elem < smallest:
      smallest = elem
  print(smallest)

def M2(lst):
  running_sum = 0
  for elem in lst:
    running_sum += 1
  print(running_sum)
```

```
def M3(lst):
  saved = lst[0]
  for elem in lst:
    if elem > saved:
      saved = elem
  print(saved)

def M4(lst):
  running_sum = 0
  for elem in lst:
    running_sum += elem
  print(running_sum)
```

A: `max`, B: `min`, C: `sum`, D: `len`