TYPE ANNOTATIONS, KEYWORDS, & CAESAR (LECTURE)

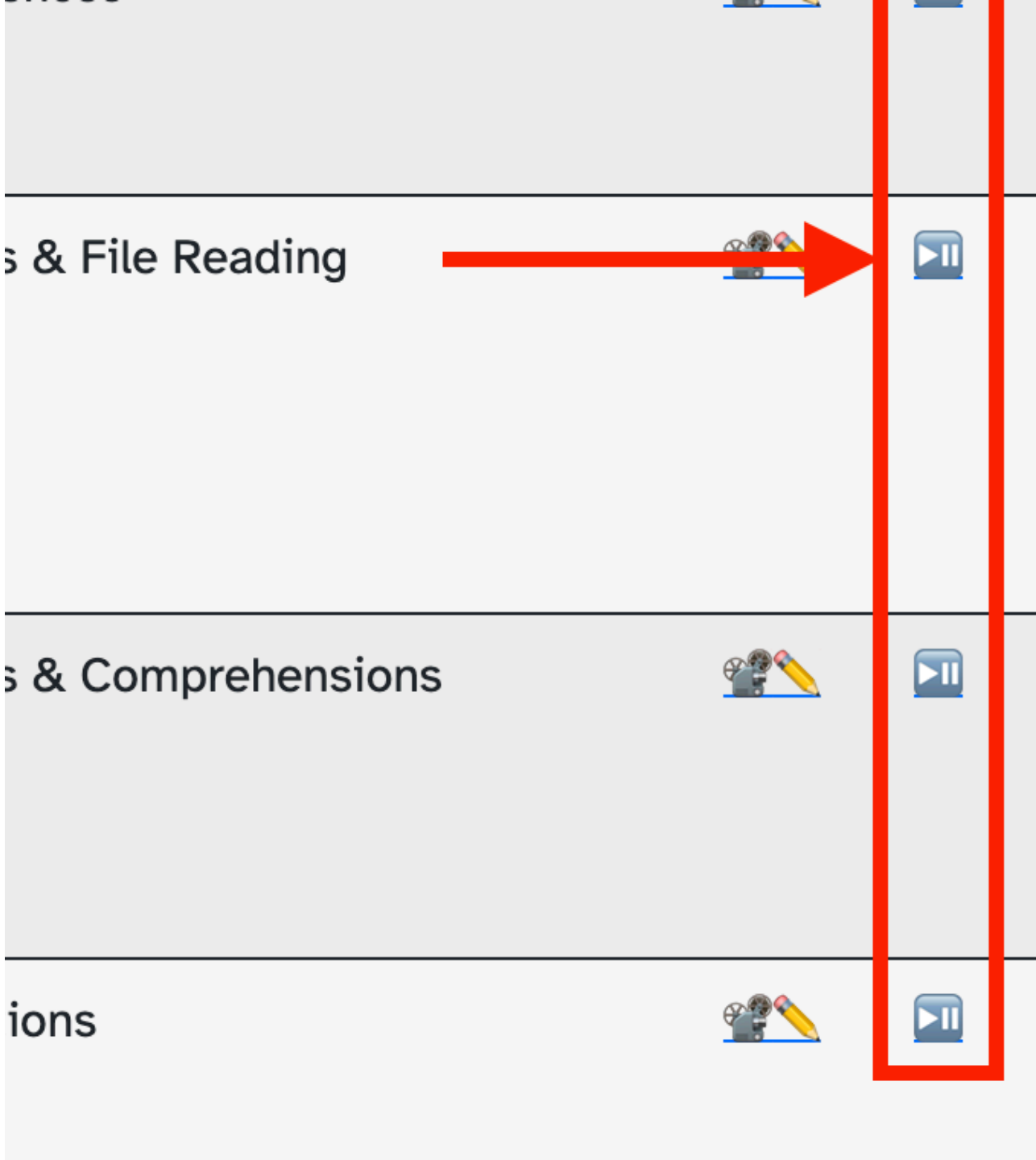
REMINDERS

- HW3 has been assigned
  - Due Weds @ 11:59PM
  - Can be submitted with late tokens until Fri @ 11:59pm
- Earn late tokens by handing in effortfully completed worksheets
- Check-in due before next class
- Sunday Review Session each Sunday 10am-noon

REMINDERS II

This course assumes that you have been watching the pre-class videos

If you have been struggling with this course and have not been watching these videos, then consider that you have been taking this course on Hard Mode



TYPE ANNOTATIONS

WHAT HAPPENS IF WE PASS WEIRD DATA INTO OUR FUNCTION?

```
def return_first_above(lst, k):
  for elem in lst:
    if elem > k:
      return elem
```

What will be returned if we call `return_first_above([1, 4, 5], "a")`?

SO HOW CAN WE NOTE WHAT OUR FUNCTION EXPECTS TO RECEIVE?

TYPE ANNOTATIONS!

```
def return_first_above(lst : list[int], k : int):
  for elem in lst:
    if elem > k:
      return elem
```

What have we changed in this function signature and what does it do?

REMINDER: SOME TYPES WE'VE SEEN

- integer -- int
- float
- string -- str
- list
- boolean -- bool

ACTIVITY: FIX THIS FUNCTION

Fill in an appropriate and descriptive docstring, and create new signature, including type annotations for the following function **L11**

```
def repeat_substring(text, substring, times):
    """
    Docstring description
    input:
    output:
    """
    result = ""
    if substring in text:
        for blah in range(times):
            result += substring
    return result
```

SO NOW WE HAVE A WAY TO TELL PEOPLE WHAT WE WANT TO TAKE IN BUT WHAT ABOUT WHAT WE'RE GIVING BACK?

WE CAN TELL USERS OUR RETURN TYPE

```
def repeat_substring(text, substring, times) -> <return_type_here>:
    result = ""
    if substring in text:
        for blah in range(times):
            result += substring
    return result
```

ACTIVITY: BE PYTHON

```
def mystery(s1: str, s2: str) -> list[str]:
    result = []
    length = len(s1)
    if len(s2) < length:
        length = len(s2)
    for i in range(length):
        result.append(s1[i])
        result.append(s2[i])
    return result
```

Which of the following is the result of the call `mystery("abcd", "qrstv")`? M3

A) `"abcdqrstv"`

B) `"aqbrcsdt"`

C) `["a", "q", "b", "r", "c", "s", "d", "t"]`

D) `'aqbrcsdtv'`

CAESAR

The next homework we will release is `Caesar`.

Warning: This homework is typically a bump up in difficulty. It is still totally doable and almost everyone finishes it. Please:

- Start early
- read the write-up
- ask for help when you need it.

This lecture will introduce a few concepts that we hope are useful to you

LET'S IMAGINE WE'RE WORKING FOR CAESAR

- Oh no! We're at war and I need to send my troops some messages! How can I do that without my enemies also reading my letters?
- Caesar what do we do?

CAESAR SHIFT

The main idea behind this is that we are interacting with a Caesar cipher.

The core idea of how this works is that we can match each letter to a number:

LETTER	A	B	C	D	E	F	G	H	I	J	K	L	M	…	X	Y	Z
Number	0	1	2	3	4	5	6	7	8	9	10	11	12	…	23	24	25

Once we do this we can do "arithmetic" on a message. e.g. `B + C -> D` and `A + B -> B`

A caesar takes a message and shifts (adds) all letters in the message by some other input letter. e.g. `CAFE` shifted by `B` is `DBGF`

CAESAR SHIFT

LETTER	A	B	C	D	E	F	G	H	I	J	K	L	M	...	X	Y	Z
Number	0	1	2	3	4	5	6	7	8	9	10	11	12	...	23	24	25

Once we do this we can do "arithmetic" on a message. e.g. `B + C -> D` and `A + B -> B`

A caesar cipher takes a message and shifts (adds) all letters in the message by some other input letter. e.g. `CAFE` shifted by `B` is `DBGF`

What is `HI` shifted by `D` ? **S7**

CAESAR SHIFT

A caesar takes a message and shifts (adds) all letters in the message by some other input letter.
e.g. `CAFE` shifted by `B` is `DBGF`

Notice how the meaning of the word afterwards is "encrypted" (the contents of the message are hidden).

Theoretically one could only get the original message if they knew how much to "subtract" from the message.

READING THE SPECIFICATION

We try to give a lot of hints in the specification, please do read the whole thing. Keep it open as you work on the assignment.

Note some of the hints we have in how we format it (Demo: `personality_quiz.py` specification)

ASCII

An important part of caesar.py is that we can convert a character to a integer using the `ord()` function.

`ord('A')` returns `65`
`ord('B')` returns `66`
`ord('C')` returns `67` etc.

We can convert an integer back to a character with `chr()`:

`chr(65)` returns `A`
`chr(66)` returns `B`
`chr(67)` returns `C` etc.

If we want to convert A to 0, B to 1, C to 2 etc. how can we do that? S8

If we want to convet 0 to A, 1 to B and 2 to C etc. how can we do that? S9

ORD PRACTICE C14

We have a list that contains some sequence of the letters `'A'`, `'B'`, `'C'`, and `'D'`. Write the function `get_counts()` that returns a list of four integers, containing the number of times `'A'`, `'B'`, `'C'`, and `'D'` show up in the input respectively. Without using any if statements.
for example:

```
letters = ['A', 'C', 'D', 'C', 'A', 'A', 'A', 'C']
counts = get_counts(letters)
print(counts) # prints [4, 0, 3, 1]
```

1. Write the function header
2. create the initial values of the result list
3. populate the result list (probably want to use `for` and `ord`)

MAIN

So far in this class we have put all our code directly in the py file. Now that we know functions, we should follow better practices.

Starting with HW03 onwards, all code should go in a function.

- import statements should still be at the top of the py file outside of functions
- Comments can be outside of functions

HELLO WORLD W/ MAIN()

From now on we will have a function called `main()` that will be where we keep the code we previously did not put in a function

We also need to add an if statement to the bottom of our code. Do not forget it!

For example:

```
def main():
    print("Hello World!")

if __name__ == "__main__":
    main()
```

MAIN() PRACTICE C16

rewrite the following code so that we use the new style that uses:

`def main():` and

`if __name__ == "__main__":`

```
import sys

# Prints all command line args
def print_args():
  for arg in sys.argv:
    print(arg)

print("Hello!")
print_args()
```

WHY MAIN()?

Writing our code in a `main()` function has a few benefits:

- Usually means our code is easier to read and better organized
- We can now import functions from the python files we write
    - this is the most important point
    - means we can import our own code into separate files for testing (or for the REPL)
- This is also just a style convention followed by most programming languages

INPUTS, PARAMETERS, ARGUMENTS

```
def add(x, b):
    return x + b

zinc = 64
add(3, zinc)
```

Previously we would call `x`, `b`, `3`, `zinc` all "inputs"

Technically we would make the distinction where
`x` and `b` are **parameters** (The variables defined in the function header)
`3` and `zinc` are **arguments** (The actual data passed into the function)

You do not need to memorize this distinction, this is just for your future use and so you know what we mean when we say parameter and argument of a function.

KEYWORD ARGUMENTS

Sometimes we want our functions to be able to take **default values** for their inputs. We can do this with *keyword arguments*.

```
def divide(a, b, rounding=False):
    result = a / b
    if rounding:
        return round(result)
    else:
        return result
```

`rounding` is a keyword argument that is defined by its **name** as well as the **default value** that it takes if it is not replaced.

KEYWORD ARGUMENNTS

```
def divide(a, b, rounding=False):
    result = a / b
    if rounding:
        return round(result)
    else:
        return result
```

We can do any of the following:

```
>>> divide(3422, 194)
17.63917525773196
>>> divide(3422, 194, rounding=True)
18
>>> divide(3422, 194, True)
18
>>> divide(3422, 194, False)
```

RULES OF KEYWORD ARGUMENTS

Signatures:

- All keyword parameters have to be provided AFTER all the positional ones
- A keyword parameter is defined by writing `identifier=<default_value>`
- Can have as many as you want, including *only* keyword parameters

Calls:

- All keyword arguments have to be passed in *after* all positional inputs, but from there can be in any order
- Keyword arguments can be given positionally or by name, but you should always just give them by name

GOOD OR BAD?

```
def fun(a, b, c=13, d):
    pass
```

GOOD OR BAD?

```
def fun(a, b, c=13, d):
    pass
```

BAD!

GOOD OR BAD?

```
def fun(a=13, n="haha"):
    pass
```

GOOD OR BAD?

```
def fun(a=13, n="haha"):
    pass
```

GOOD!

GOOD OR BAD?

```
def fun(a, b, c=, d=13):
    pass
```

GOOD OR BAD?

```
def fun(a, b, c=, d=13):
    pass
```

BAD!

GOOD OR BAD?

```
def fun(x, y, z=0):
    pass
```

then,

```
...
fun(3, 4, 0)
...
```

GOOD OR BAD?

```
def fun(x, y, z=0):
    pass
```

then,

```
...
fun(3, 4, 0)
...
```

OK, but redundant?

GOOD OR BAD?

```
def fun(x, y, z=0):
    pass
```

then,

```
...
fun(z=0, 3, 4)
...
```

GOOD OR BAD?

```
def fun(x, y, z=0):
    pass
```

then,

```
...
fun(z=0, 3, 4)
...
```

BAD!

GOOD OR BAD?

```
def fun(x, y, z=0):
    pass
```

then,

```
...
fun(3, 4, z=x+y)
...
```

GOOD OR BAD?

```
def fun(x, y, z=0):
    pass
```

then,

```
...
fun(3, 4, z=x+y)
...
```

BAD!

GOOD OR BAD?

```
def fun(x, y, z=0):
    pass
```

then,

```
...
fun(3, 4)
...
```

GOOD OR BAD?

```
def fun(x, y, z=0):
    pass
```

then,

```
...
fun(3, 4)
...
```

Good!