Warmup (L8)

Write a function that computes pi using the Leibniz formula, taking a callback to decide when to stop. The callback should be a function that takes a float (the current approximation) and returns True to indicate “stop now”, False otherwise.

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \cdots$$

Prime factorization generalized

Let’s generalize our prime factorization function to do anything (rather than just `print`) for each factor

Prime factorization generalized

```
import typing

def primeFactors(n: int, cb: typing.Callable) -> None:
    assert n > 0, "Only positive integers
                   have factors"
    least = 2
    while n > 1:
        f = least
        while f < n and n%f != 0:
            f = f + 1
        cb(f)
        least = f
        n = n // f

def printAsFloat(x: int) -> None:
    print(float(x))

primeFactors(42, printAsFloat)
```

Prime factorization generalized

```
import typing

def primeFactors(n: int, cb: typing.Callable) -> None:
    assert n > 0, "Only positive integers
                   have factors"
    least = 2
```

“`cb`” (for “callback”) is a common name for a function argument when there’s no descriptive name for it

```
            cb(f)
            least = f
            n = n // f

def printAsFloat(x: int) -> None:
    print(float(x))

primeFactors(42, printAsFloat)
```

Debugging generalized

- It’s common to enable or disable debugging prints globally instead of commenting out each one
- How do we do that? By storing `print` in a variable, then changing it when we don’t want to print!
- But changing it to what…

Mocks

- Python provides a “don’t do anything” function (mainly for testing):

```
from unittest.mock import Mock
doNothing = Mock()
doNothing()  # Does nothing
```

Note that `Mock` is a function that returns a function!
Make sure to call it!

Debugging generalized

```python
from unittest.mock import Mock

debug = print

def sqrtButTerrible(n: float) -> float:
    r = n / 2
    debug("Initial guess:", r)
    while abs(r**2 - n) >= 0.0001:
        r = (r + n/r) / 2
        debug("Guess in loop:", r)
    debug("Final value:", r)
    return r
```

Debugging generalized

```python
from unittest.mock import Mock

debug = Mock() # One change, prints go away!

def sqrtButTerrible(n: float) -> float:
    r = n / 2
    debug("Initial guess:", r)
    while abs(r**2 - n) >= 0.0001:
        r = (r + n/r) / 2
        debug("Guess in loop:", r)
    debug("Final value:", r)
    return r
```

More examples

- Let’s do some more examples using loops:
  - Compute compound interest
  - Compute pi using the Leibniz formula

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \cdots$$

Module summary

CS114 M3

Module summary

- You’ve seen how to repeat in your code with **`while`** and **`for`** loops
- **`while`** loops can have sophisticated conditions
- Sometimes the condition is about when it ends, sometimes when it continues
- **`for`** loops can use ranges
- Abstraction inverted: functions are values

CS114

Module 4: Strings and Lists

Sequences

CS114 M4

Sequences

- We discussed ranges for **`for`** loops
- I said it’s a “grouping”, but it’s more specific: a *sequence*
- A sequence is a grouping of items with some order
  - `range(1, 10)`: 1, 2, 3, 4, 5, 6, 7, 8, 9
  - prime numbers: 2, 3, 5, 7, 11, …

We’ve already seen a sequence!

- Strings are just sequences of characters! (“Character” is a general term for a glyph used in language)
    - You could say the characters have been *strung* together. Yup, that’s the etymology.

```
for c in "Hello, world!":
    print(c)
```

Manipulating sequences

- As we've seen, we can loop over sequences
- We can also get elements from sequences by *indexing*

```
print("Hello, world!"[1])
e
print(range(1, 10)[2])
3
```

Manipulating sequences

```
print("Hello, world!"[1])
e
print(range(1, 10)[2])
3
```

(some sequence)[*index*] gets an element from a sequence

Manipulating sequences

```
print("Hello, world!"[1])
e
print(range(1, 10)[2])
3
```

Surprised by the results?
Sequences in Python (and most programming languages) are *0-indexed*. That means that the index for the first element is 0, not 1.

Aside on 0-indexing

- A common error is the *off-by-one* error, which is exactly what it sounds like
- Some people think 0-indexing is the cause of off-by-one errors
- When Julius Caesar was assassinated, Julian leap years were done wrongly for 50(ish) years due to an off-by-one error. Humans just suck at counting.

Sequence length

- Get the length of any sequence with `len`
- We can use ranges to loop over elements in a different way:

```
s = "Hello, world!"
for i in range(len(s)):
    print(s[i])
```

Modifying sequences

- You can *access* the individual characters in a string, but you can’t *change* them

```
x = "Hello, world!"
x[1] = "u" # ERROR!
```

- strings are *immutable* (un-changeable)
- So are ranges

Lists

CS114 M4

Lists

- Lists are sequences that can contain anything
- Written with square brackets:

```
[2, 4, 6, 0, 1]
```

- Indexed like any sequence

```
x = [2, 4, 6, 0, 1]
x[2] == 6
```

Typing lists

- The type for a list is `list`
- But most of the time, you care what it's a list *of!*
- You can specify what's in the list with, e.g., `list[int]`
- It is always the right style to type as specifically as possible. Don't use `list` when you know what's in it!

List example

- Let’s write a function to average a list of numbers

List example

- Let’s write a function to average a list of numbers

```
def averageOf(l: list[float]) -> float:
    sum = 0.0
    for val in l:
        sum = sum + val
    return sum / len(l)

averageOf([2, 4, 6, 0, 1]) # 2.6
```

In-lecture quiz (L8)

- https://student.cs.uwaterloo.ca/~cs114/F25/quiz/
- Q1: How many times does this print "x"?

```
for s in ["Excellent", "text", "box"]:
    for c in s:
        print(c)
```

A. 0 (no times)

B. 1

C. 2

D. 3

E. 4

In-lecture quiz (L8)

- https://student.cs.uwaterloo.ca/~cs114/F25/quiz/
- Q2: What does this print?

```
print(len(["Excellent", "text", "box"]))
```

A. Nothing or an error
B. Excellent text box
C. 3
D. 16
E. 18

List example

- Let’s write a function to check if a value is in a ~~list~~ sequence (any type of sequence!)

List example

- Let’s write a function to check if a value is in a ~~list~~ sequence (any type of sequence!)

```
def contains(
  haystack: typing.Sequence,
  needle: typing.Any
) -> bool:
    for val in haystack:
        if val == needle:
            return True
    return False
```

List example

```
def contains(
  haystack: typing.Sequence,
  needle: typing.Any
) -> bool:
            return True
    return False
```

The type for a sequence of any sort (string, list, range) is in the typing module.

List example

```
def contains(
  haystack: typing.Sequence,
  needle: typing.Any
) -> bool:
    for val in haystack:
    return False
```

This type means “I don’t care”. In this case, we’re not *doing* anything with the needle, so we don’t actually care what it is.

List example

```
def contains(
   haystack: typing.Sequence,
   needle: typing.Any
) -> bool:
     for val in haystack:
     return False
```

Be wary of this type!
Remember: types are documentation! Don’t just write “any” to make the type checker shut up!

The `in` operator

- We just wrote a `contains` function
- As it turns out, Python has this built in:

```
x = [2, 4, 6, 0, 1]
6 in x # True
"e" in "hello" # True
```

Lists are mutable

- Unlike the other sequences we’ve seen so far, lists are *mutable* (changeable)

```
x = [2, 4, 6, 0, 1]
print(x) # [2, 4, 6, 0, 1]
x[1] = 8 # change an element just
         # like you’d change a
         # variable
print(x) # [2, 8, 6, 0, 1]
```

Using mutation

- Let’s replace every value in a list with the running average (the average until that point in the list)

```
def runningAverage(l: list[float]) -> float:
    sum = 0.0
    for idx in range(len(l)):
        sum = sum + l[idx]
        l[idx] = sum / (idx+1) # 0-indexing!
    return sum / len(l)
```

Using mutation

- Let’s replace every value in a list with the running average (the average until that point in the list)

Values in the list are replaced (after we used them)

```
def runningAverage(l: list[float]) -> float:
    sum = 0.0
    for idx in range(len(l)):
        sum = sum + l[idx]
        l[idx] = sum / (idx+1) # 0-indexing!
    return sum / len(l)
```

Modeling memory

CS114 M4

How data is stored

- The association of variable names with values is part of the *memory* of the computer
- Each variable is said to have a *slot* in memory that stores a value
- With mutable lists, we’ll find that the arrangement of memory is complicated!
- We need a mental model of how memory works

Why it’s hard

```
x = [2, 4, 6, 0, 1]
y = x
x[1] = 8
print(y[1]) # prints 8
for i in y:
    i = 0
print(y[1]) # prints 8
```

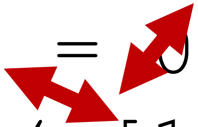
Why it’s hard

```
x = [2, 4, 6, 0, 1]
y = x
x[1] = 8
print(y[1]) # prints 8
for i in y:
    i = 0
print(y[1]) # prints 8
```

A change in `x` was visible in `y`

Why it’s hard

```
x = [2, 4, 6, 0, 1]
y = x
x[1] = 8
print(y[1]) # prints 8
for i in y:
    i = 0
print(y[1]) # prints 8
```



And yet this changed nothing!

The graph model of memory