Today
Lazy lists
Side effects

Looking at creative uses of
"functions of data"

Lazy evaluation - only generate
data when you need it
Typically, we code w/ "eager evaluation"
- run your code right now, when
you call your functions

Paradigms of programming
- side effects

Major programming paradigms

Imperative programming -
- series of operations modifying machine state as you go
- most of what you've done
  most of C, some of Python, Java, etc

Object-oriented programming
- operations modifying objects
- OO parts of Python, Java, Kotlin, etc.

Logical programming
- program is a collection of facts and rules, and the program makes logical deductions
  Prolog, libraries for major languages that do this stuff

Functional programming
- everything is a function call
Scheme, Haskell, parts of Python,
Java, Kotlin, etc.

One main reason this model is useful is it can make debugging dramatically easier if you have to reason about all the wacky things your functions might be changing

In pure functional programming, there are no side-effect

A function call, or any expression has a side-effect if it makes any kind of observable change or interaction, beyond returning a value.

```
def inc(x):
  return x+1
```

```
   :
inc(3) → 4
```

There are no side-effects when I call inc(3)

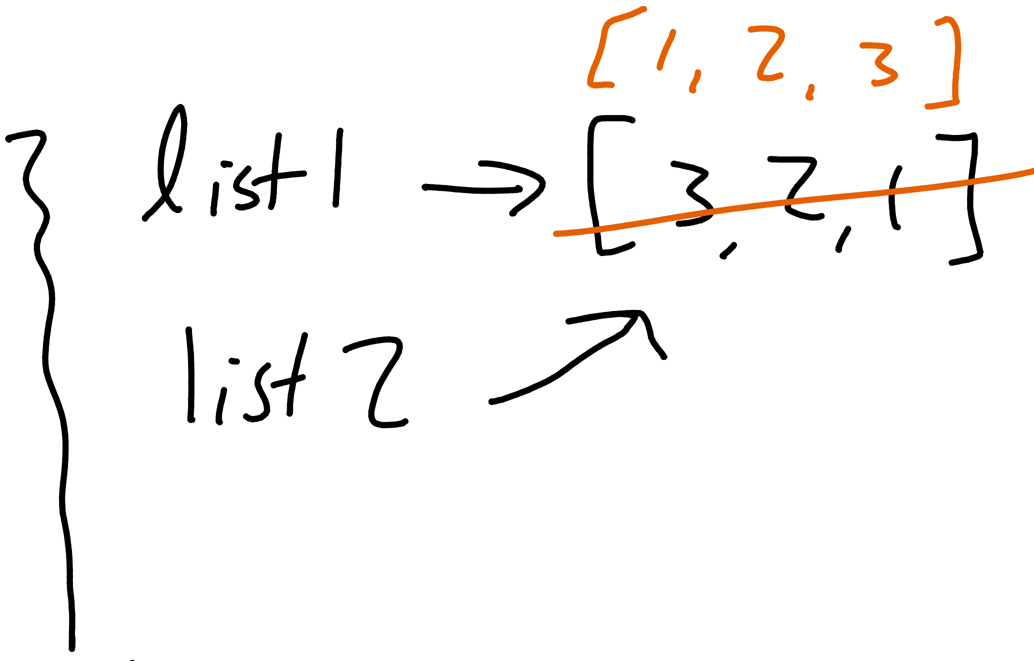
```
print("hello")
```

returns None, but has a side-effect of displaying "hello" in the terminal.

What are examples where side-effects can be problematic?

Example 1:

```
list1 = [3, 2, 1]
list2 = list1
# millions of lines of code
# switch developers
list2.sort()
print(list1)
```



→ side effect of modifying object might be viewed unexpectedly

↳ and programmer is shocked to discover it prints [1, 2, 3]

Example 2:

(see code) example.py

if you are not (potentially) modifying things, there's no confusion about what you might have modified.

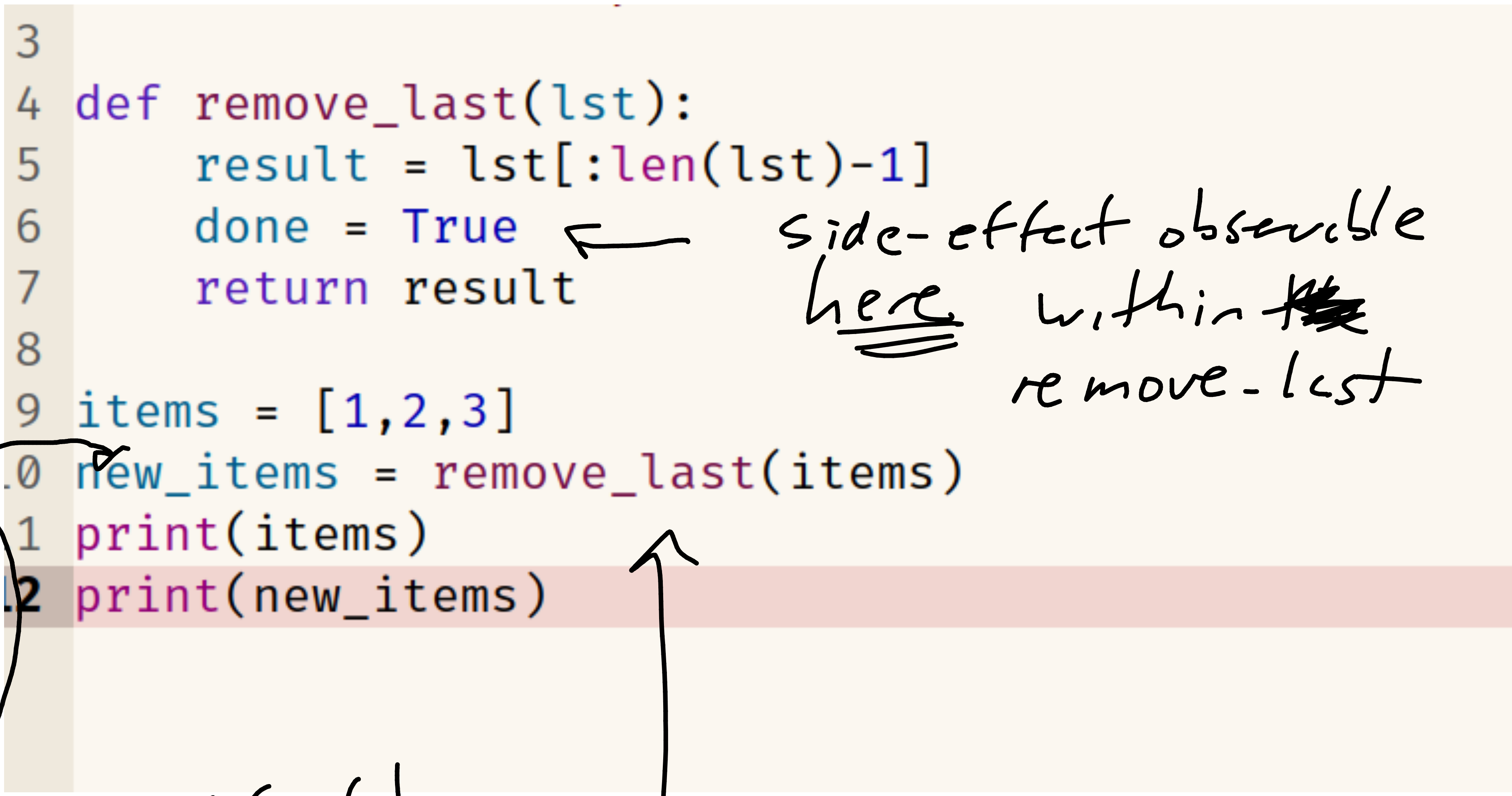
Example 3: (concept, not actual Python)

```
x = 3

def doit1():
    global x
    x = 8

def doit2():
    global x
    x = 10

#doesn't exist, but does in concept
do-in-parallel(doit1, doit2)
print(x)
```

never any guarantee on precise timing when things are running simultaneously

I don't know which x I will get.

If there are no side-effects, it doesn't matter what order they run in



```python

def remove_last(lst):
    result = lst[:len(lst)-1]
    done = True
    return result

items = [1,2,3]
new_items = remove_last(items)
print(items)
print(new_items)
```

side-effect observable here within ~~the~~ remove_last

side effect!

When I call remove_last, no observable changes happen from the pespective of the caller of remove_last.

"Calling remove_last results in no side-effects." Accurate.

But done = True modifies a variable done.