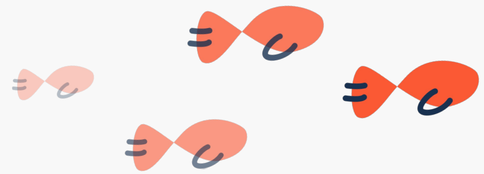
neue fische

School and Pool for Digital Talent

Revisiting Python Fundamentals



Beginner Workshop

Larissa Hubschneider

- Biotechnologist
- Coach Data Science Bootcamp
- love to knit socks for other people in winter

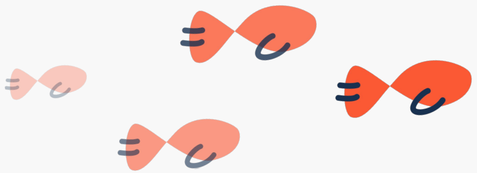
Beginner Workshop

Nico Steffen

- Naval Architect R&D Engineer
- Coach Data Science Bootcamp
- would like to get socks knitted in winter

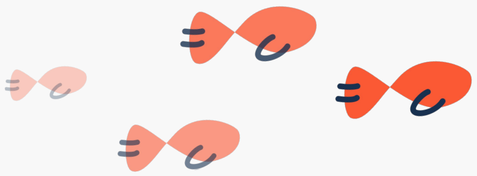
neuefische - School and Pool for Digital Talent

- founded 2018 by Dalia Das
- Large alumni and partner network
- 12 Week Bootcamps:
  - Web Development
  - Data Science
  - Data Analytics
  - Java Development
  - Cloud Development



neuefische - upcoming Data Science Bootcamps

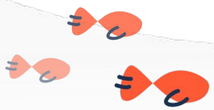
- Data Science Bootcamps:
  - 13.9 and 27.9
- for more details of our Bootcamps, checkout our website:
  - neuefische.de
- or get in touch with us:
  - studienberatung@neuefische.de



Beginner Workshop

Revisiting Python Fundamentals

- About: Who are we?
- Revisiting 1: List comprehension
- Revisiting 2: Copying lists in Python
- Revisiting 3: Generators
- Q&A



How to use python list comprehension
Or: The calendar Incident

Yay! Python 3!

... you all know lists.

... you all know several ways to create lists

but do you know enough about list comprehensions?

... psst. psst. you, yes you!
... are you looking for some information? about leap-years? I can hook you up!

Fundamentals 1: List comprehension

Q: “ Use python to create a list of the first 100 numbers”

Answer 1

```
first_numbers=[  0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
                10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
                20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
                30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
                40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
                50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
                60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
                70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
                80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
                90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
```

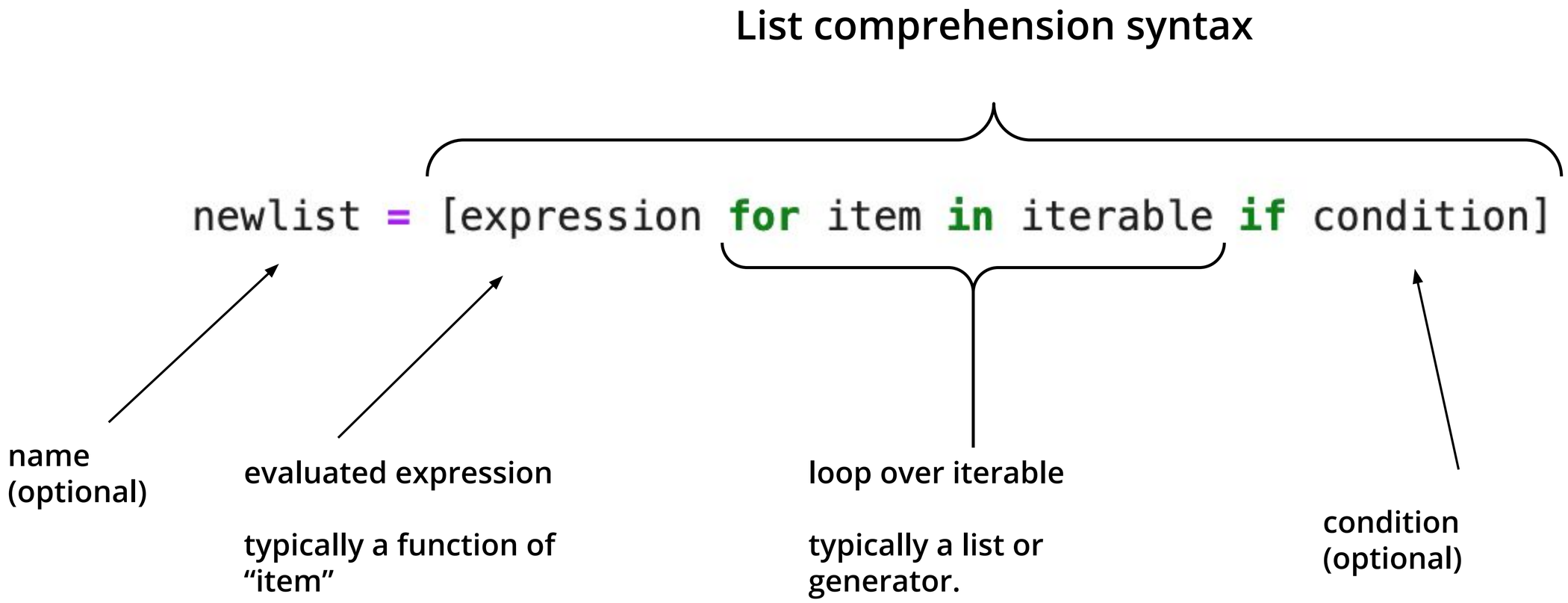
Answer 2

```
first_numbers=[]
for i in range(100):
    first_numbers.append(i)
```

Answer 3

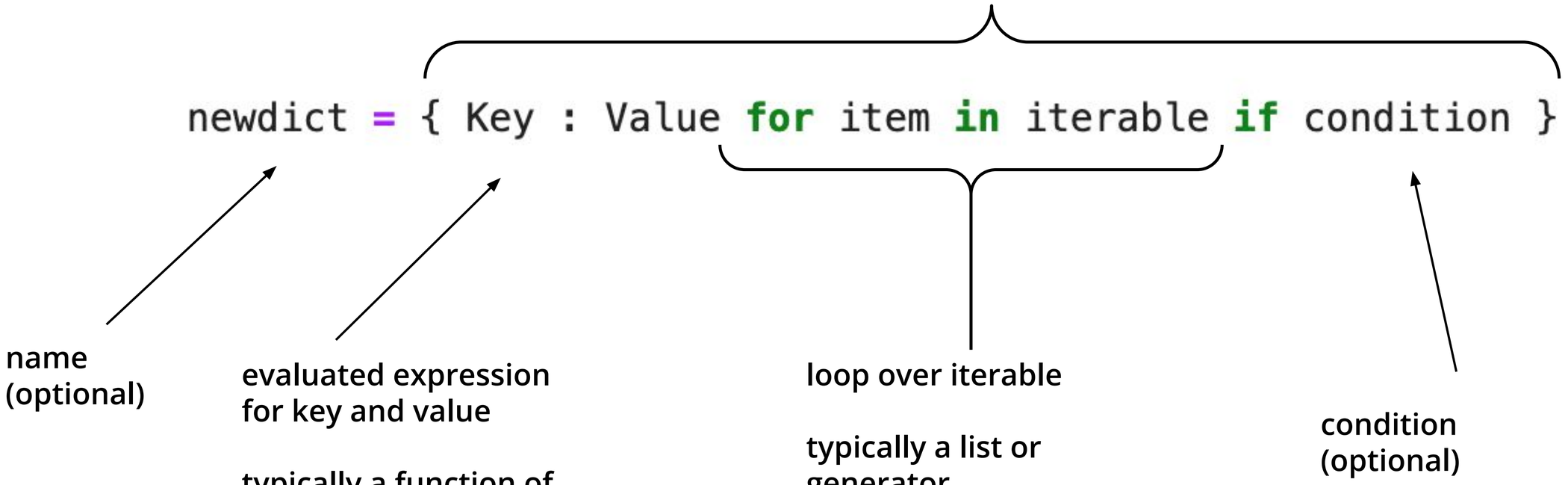
```
first_numbers=[i for i in range(100)]
```

Fundamentals 1: List comprehension



Fundamentals 1: List comprehension

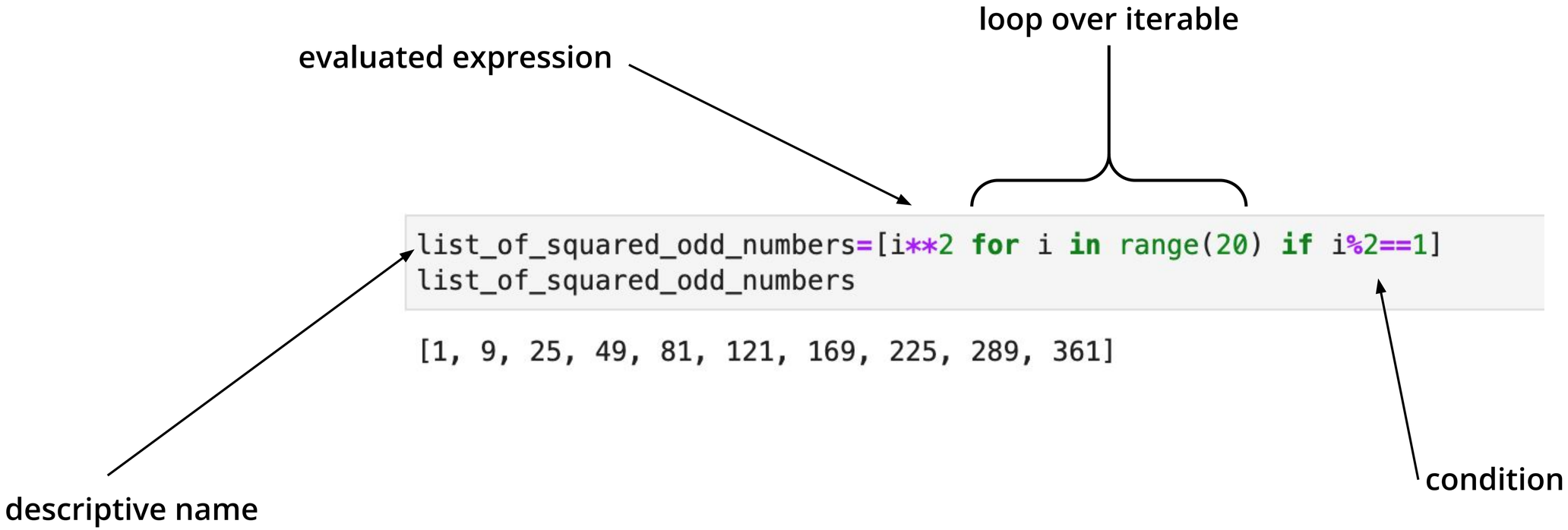
Dictionary comprehension syntax



Fundamentals 1: List comprehension

1. Example

   Square all odd numbers smaller than 20 and store them in a list

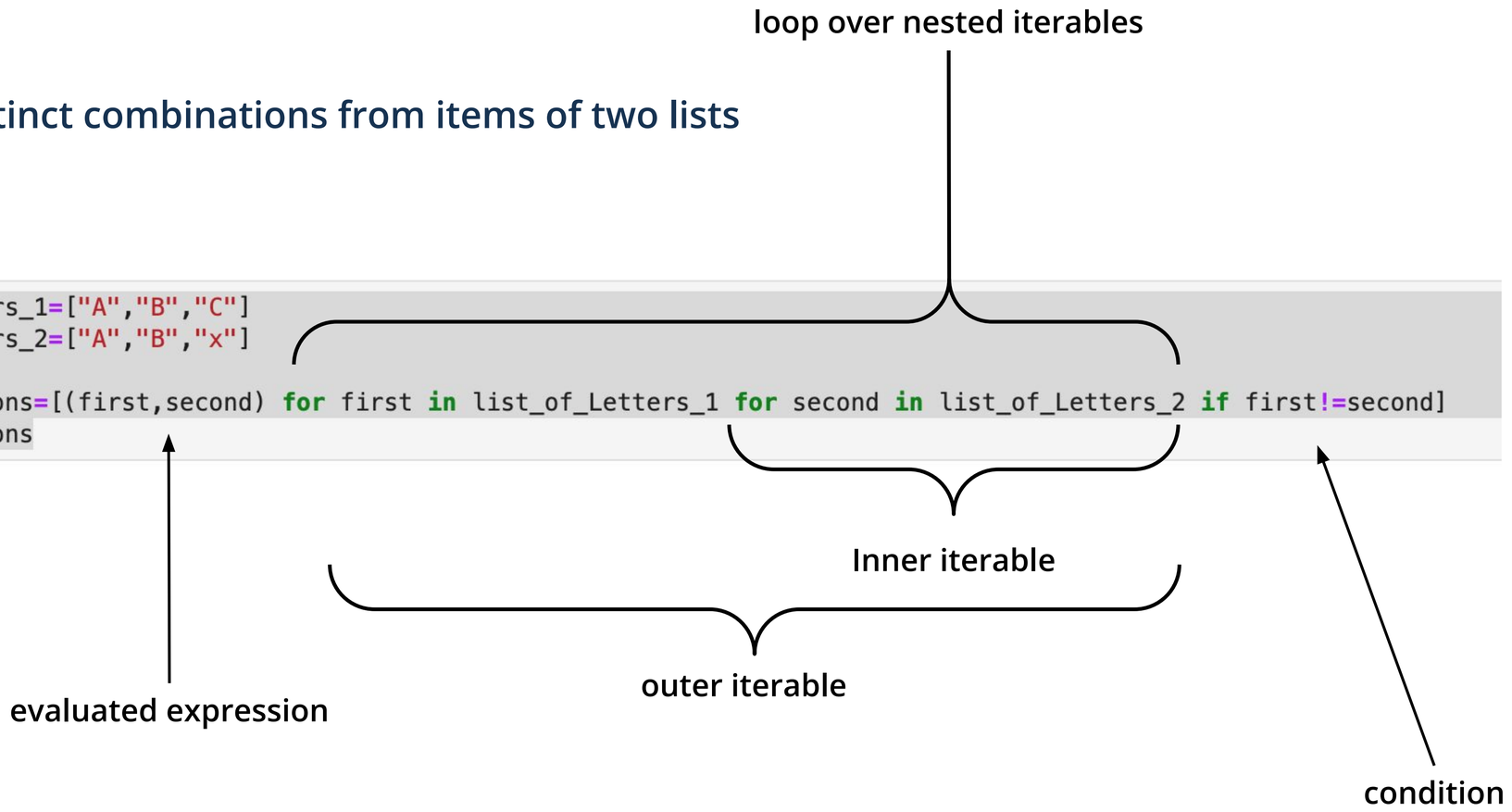


Fundamentals 1: List comprehension

2. Example
   Generate list of distinct combinations from items of two lists

```
list_of_Letters_1=["A","B","C"]
list_of_Letters_2=["A","B","x"]

all_combinations=[(first,second) for first in list_of_Letters_1 for second in list_of_Letters_2 if first!=second]
all_combinations
```

```
[('A', 'B'),
 ('A', 'x'),
 ('B', 'A'),
 ('B', 'x'),
 ('C', 'A'),
 ('C', 'B'),
 ('C', 'x')]
```



Fundamentals 1: List comprehension

3. Example
   Generate a specific string that contains the correct weekday and date for all days up to today

Bad Example

```
days=["Thu","Fri","Sat","Sun","Mon","Tue","Wed"]
months={"Jan":31,"Feb":28,"Mar":31,"Apr":30,"May":31,"Jun":30,"Jul":31,"Aug":31,"Sep":30,"Oct":31,"Nov":30,"Dec":31}
years={yr : is_leap_year(yr) for yr in range(1970,2022)}

fancy_cal=[f"{days[(ind)%7]}, {D}th of {M} {Y}" for ind,(D,M,Y) in enumerate([(D, month, year) for year,leap in years.items()
                                                                                for month,max_days in months.items()
                                                                                for D in range(1,max_days+1+(leap and (month=="Feb")))])]


p_say(fancy_cal[57:61])
p_say(fancy_cal[364*2+59:364*2+63])
p_say(fancy_cal[-121])
```

```
Python:  ['Fri, 27th of Feb 1970', 'Sat, 28th of Feb 1970', 'Sun, 1th of Mar 1970', 'Mon, 2th of Mar 1970']
Python:  ['Sun, 27th of Feb 1972', 'Mon, 28th of Feb 1972', 'Tue, 29th of Feb 1972', 'Wed, 1th of Mar 1972']
Python:  Thu, 2th of Sep 2021
```

unclear logic

unintuitive unpacking

several nested loops

hidden logic

It’s working just fine… but is it easy to read and understand? Is it easy to maintain?

Probably not.

```
fancy_cal=[f"{days[(ind)%7]}, {D}th of {M} {Y}" for ind,(D,M,Y) in enumerate([(D, month, year) for year,leap in years.items()
                                                                                for month,max_days in months.items()
                                                                                for D in range(1,max_days+1+(leap and (month=="Feb")))])]
```

Fundamentals 1: List comprehension

Recap!

Syntax

```
newlist = [expression for item in iterable if condition]
```

Be Pythonic

Python loves iterables.
And Python developers love to avoid explicit loops.

Cleaner Code

Typically when you prepare simple lists for further usage, you want to have it short and compact to not clutter your code.

Efficiency

Often List comprehensions are significantly faster than other approaches

```
def list_long():
    first_list=[]
    for i in range(10000000):
        first_list.append(i)
    return first_list

def list_comp():
    return [i for i in range(10000000)]

print(f"Duration with a loop and list.append(): {round(np.mean(timeit.repeat(list_long,number=3)),2)}s")
print(f"Duration with a list comprehension:     {round(np.mean(timeit.repeat(list_comp,number=3)),2)}s")
print(f"Are both lists identical? {'yes.' if list_long()==list_comp() else 'nope.'}")
```

✓ 10.2s

```
Duration with a loop and list.append(): 1.23s
Duration with a list comprehension:     0.66s
Are both lists identical? yes.
```

So let’s visit the notebook!!

How to copy lists in Python.
Or: The aquarium Incident

… you all know lists.

… you all know several ways to create lists including list comprehensions.

… you all know how to copy them?

but do you know that there are 3 distinct ways to copy lists?

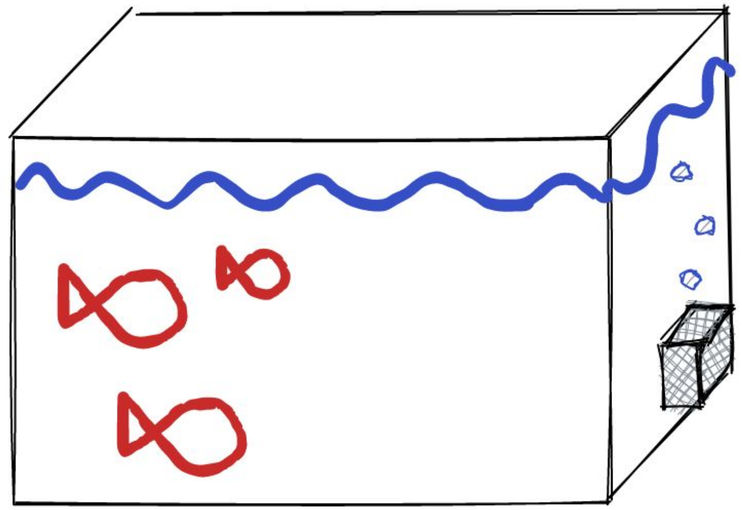
With quite different outcomes

Let’s go on a journey to my younger me…

How to copy lists in Python.
Or: The aquarium Incident

This story is all about an aquarium.
Aaand... sibling love ❤️

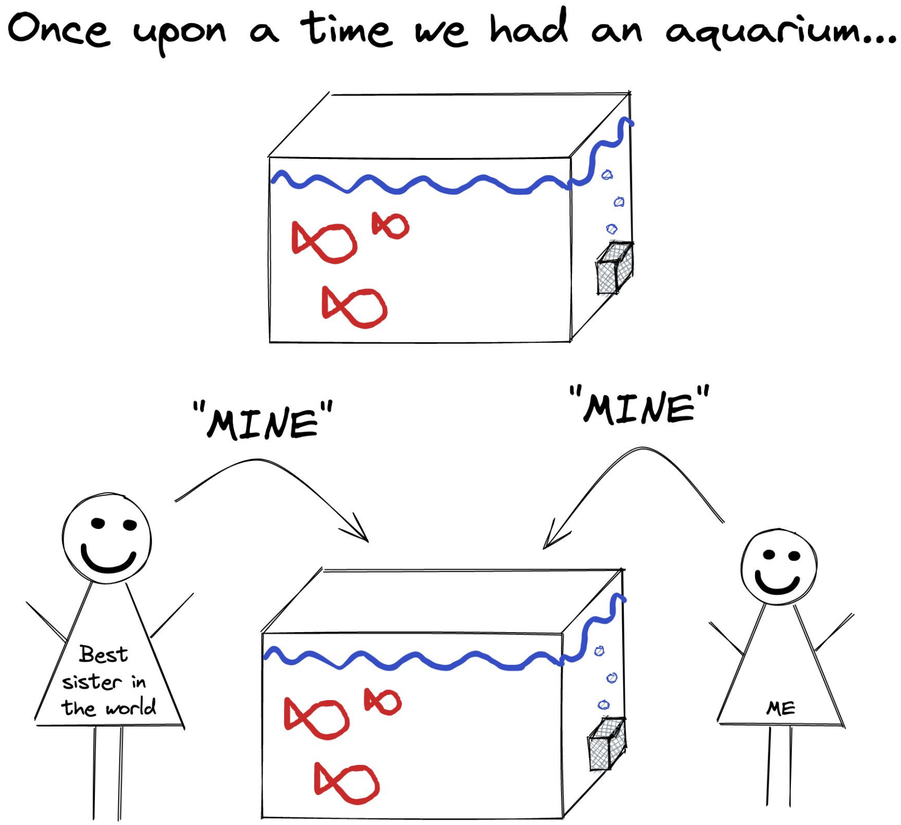
```python
# in python it's just a list
aquarium = ['blackbox', 'big fish', 'small fish', 'second big fish']
```



Chapter 1: the ‘=’ operator

We had an aquarium together.
But of course, both of us said it’s “Mine”.

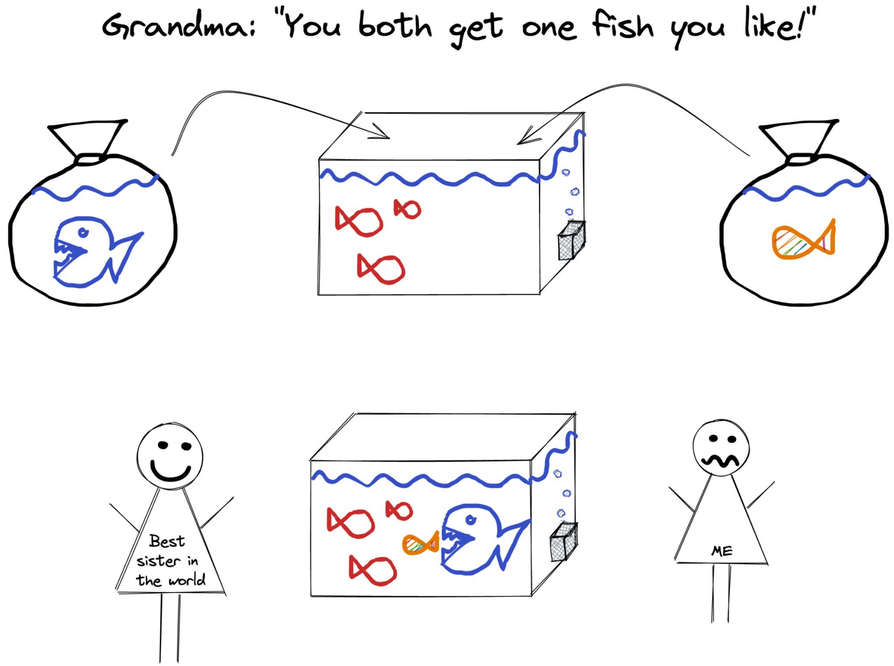
```
aquarium = ['blackbox', 'big fish', 'small fish', 'second big fish']
# using the '=' operator
my_aquarium = my_sis_aquarium = aquarium
```



Chapter 1: the ‘=’ operator

We both got fishes.
And the landed in the same aquarium!

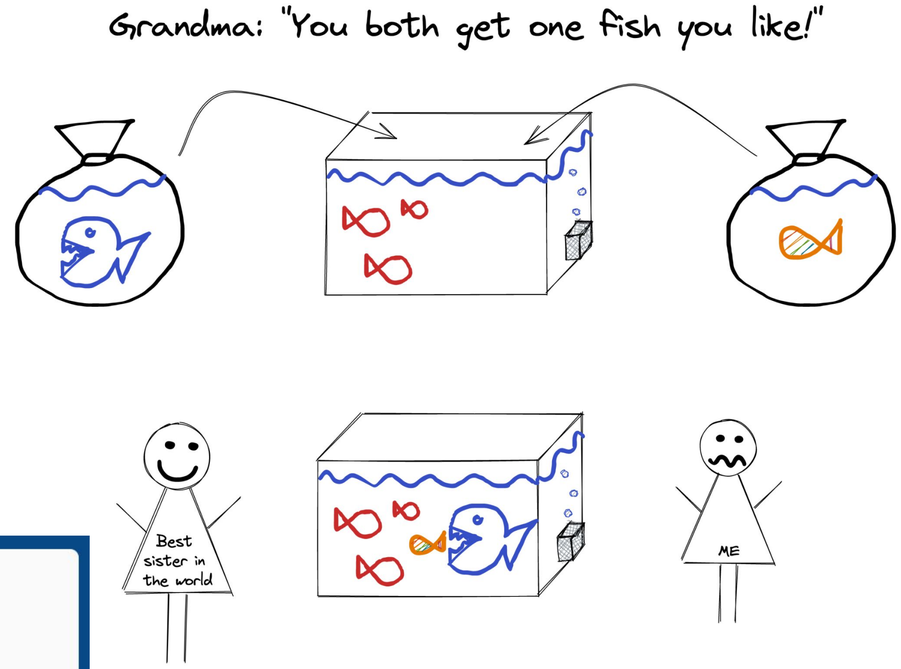
```
# Grandmas presents
my_sis_aquarium.append('Shark')
my_aquarium.append("Rainbow fish")
```



Chapter 1: the ‘=’ operator

We both got fishes.
And the landed in the same aquarium!

```
# Grandmas presents
my_sis_aquarium.append('Shark')
my_aquarium.append("Rainbow fish")
```



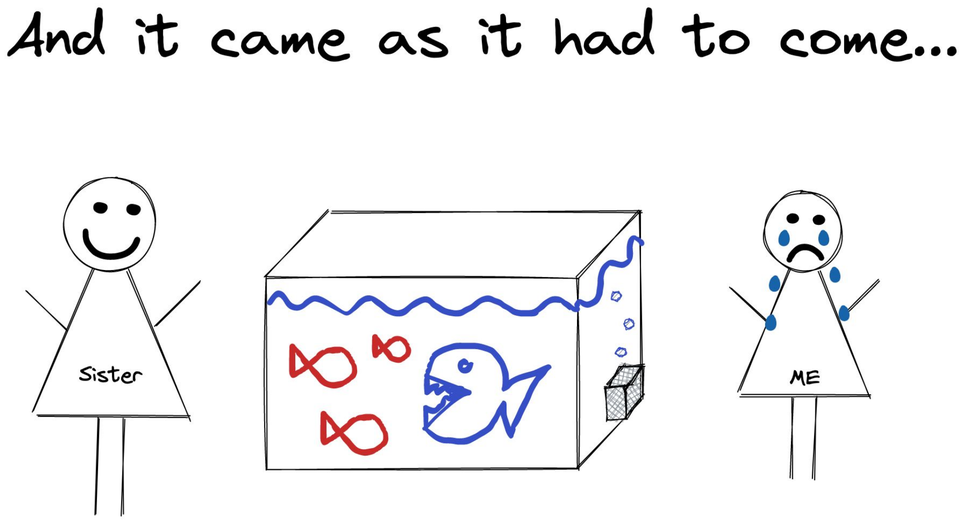
Yes! It happened exactly like I said!

```
print(my_aquarium)
>>>['blackbox', 'big fish', 'small fish',
'second big fish', 'Shark', 'Rainbow fish']
print(my_sis_aquarium)
>>>['blackbox', 'big fish', 'small fish',
'second big fish', 'Shark', 'Rainbow fish']
```

Chapter 1: the '=' operator

The shark ate my fish :(

```
# the shark ate my fish
my_sis_aquarium.pop()
>>>'Rainbow fish'
```



So, I did the only logical thing: I destroyed the aquarium!

Chapter 1: the ‘=’ operator - What happened?

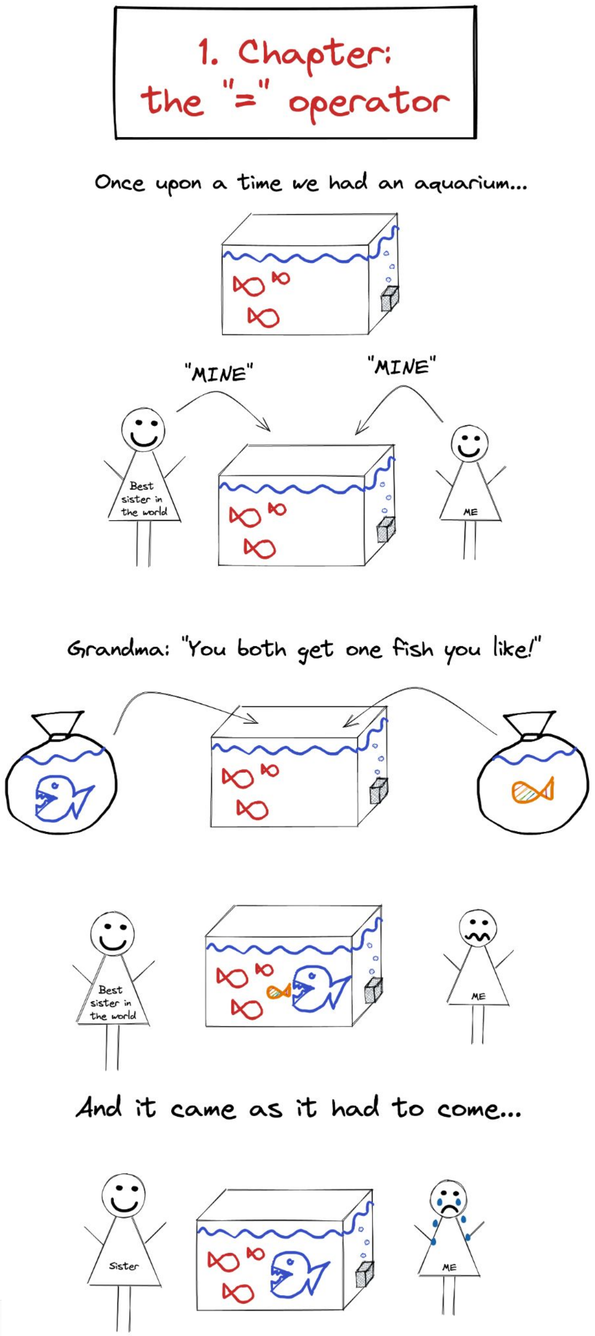
The ‘=’ operator creates a copy of an object.

But it doesn’t create a new object,
it only creates a new variable that **shares the reference to the original object**.

Like my sister and me. It was “her” or “mine” aquarium. But there was still just this 1 aquarium we referred to.

You can check this in python by looking at the id of an object.

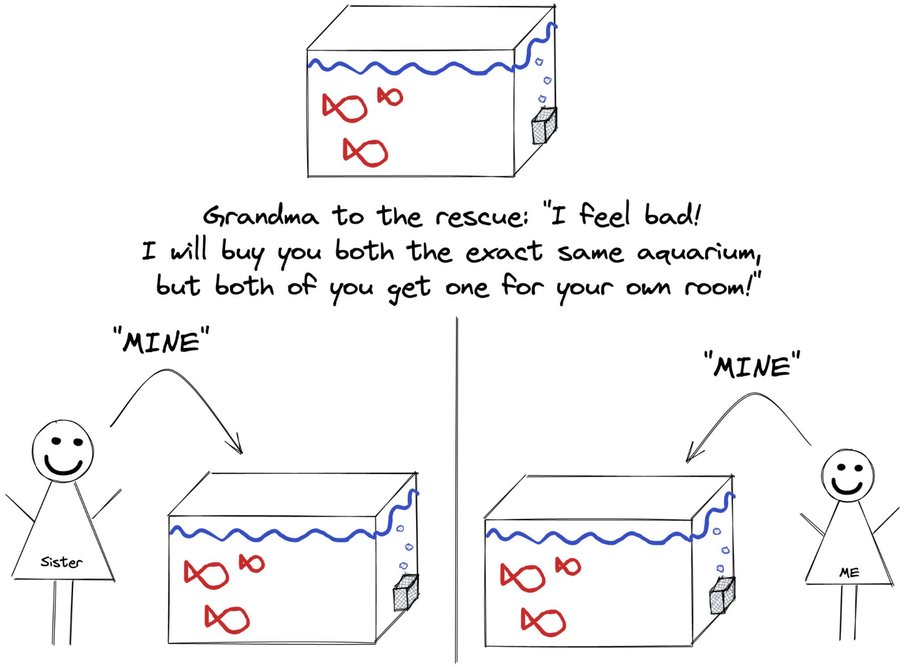
```
# the two variables refer to the same object
id(my_aquarium) == id(my_sis_aquarium)
>>>True
```



Chapter 2: Shallow Copy

This one aquarium thing didn't work out.
So we got both our own.

```
# the shallow copy
blackbox = ['20 % O2', 'Off']
my_sis_aquarium_2 = [blackbox, 'big fish', 'small fish', 'second big fish']
my_aquarium_2 = copy.copy(my_sis_aquarium_2)
```



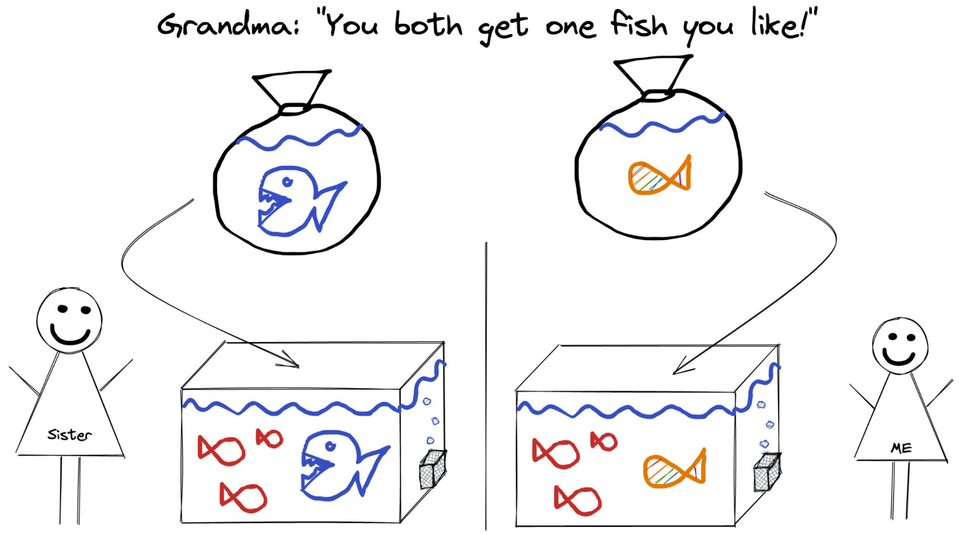
Btw. There are multiple ways in python to get to this so-called shallow copy:

```
# the "copy" method
my_aquarium_2 = copy.copy(my_sis_aquarium_2)
# the "list" method
my_aquarium_2 = list(my_sis_aquarium_2)
# the "slicing" method
my_aquarium_2 = my_sis_aquarium_2[:]
```

Chapter 2: Shallow Copy

Let’s try appending again…

```
# appending to shallow copies
my_sis_aquarium_2.append('Shark')
my_aquarium_2.append('Rainbow fish')
```



And it worked: changes in my aquarium didn’t effect the one of my sister

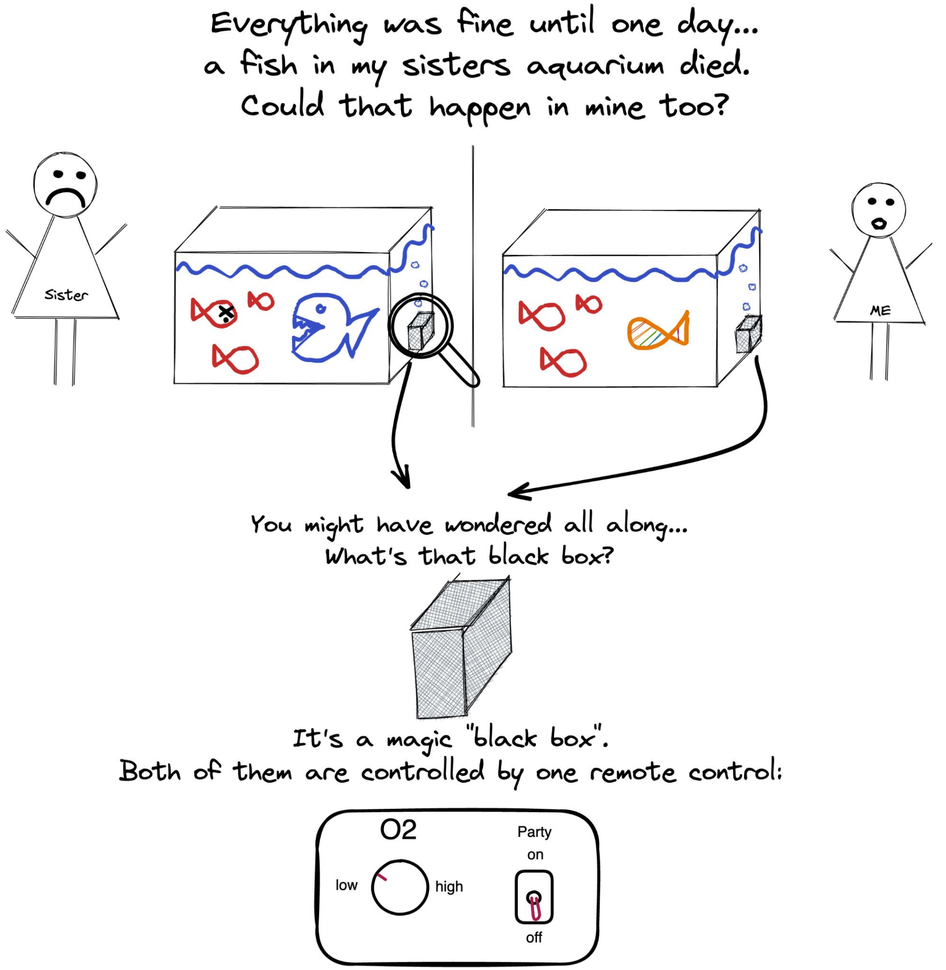
```
# appending to shallow copies
print(my_sis_aquarium_2)
>>>[['20 % O2', 'Off'], 'big fish', 'small
fish', 'second big fish', 'Shark']
print(my_aquarium_2)
>>>[['20 % O2', 'Off'], 'big fish', 'small
fish', 'second big fish', 'Rainbow fish']
```

Chapter 2: Shallow Copy

A fish died in my sisters aquarium...

```
# removing object in shallow copy
my_sis_aquarium_2.pop(-2)
```

This shouldn't happen in mine! So... let's increase O2 concentration!

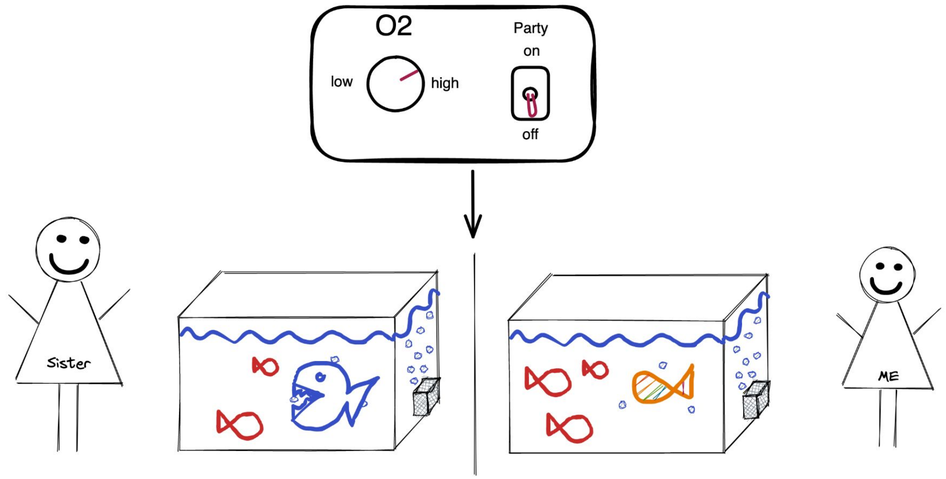


Chapter 2: Shallow Copy

Both blackboxes are controlled together!

```
# changing nested lists
blackbox[0] = "50 % O2"
print(my_sis_aquarium_2)
>>>[['50 % O2', 'Off'], 'big fish', 'small fish',
'second big fish', 'Shark']
print(my_aquarium_2)
>>>[['50 % O2', 'Off'], 'big fish', 'small fish',
'second big fish', 'Rainbow fish']
```

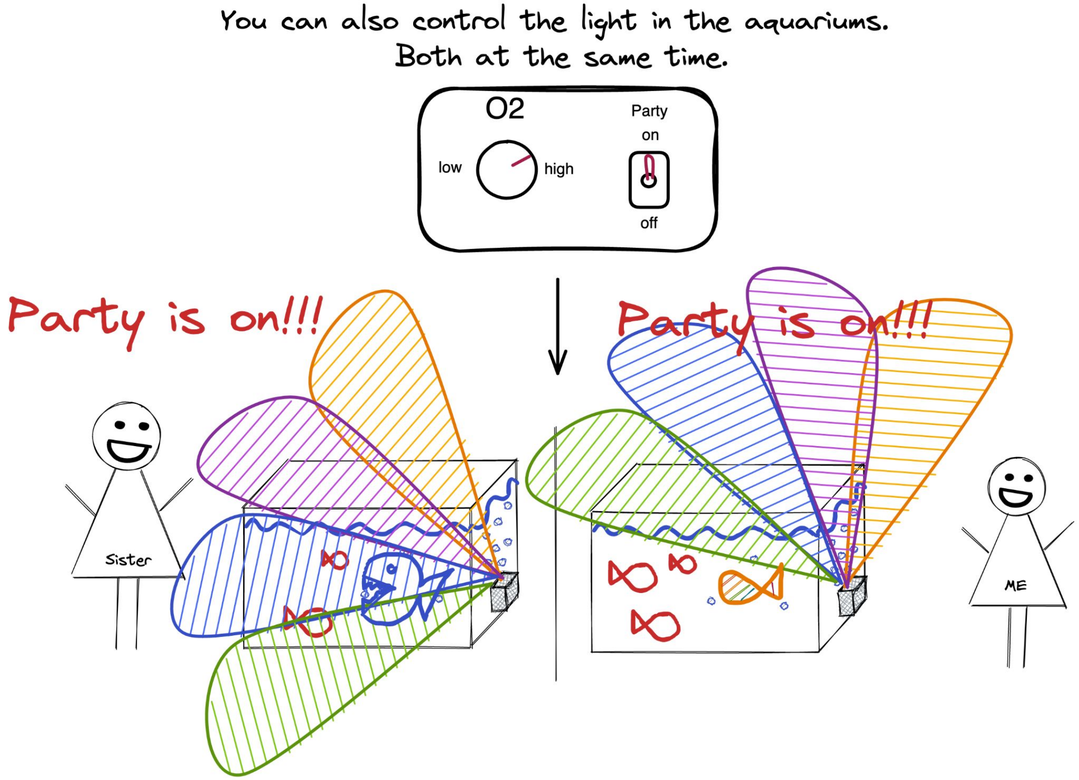
With the remote control you can increase O2 to prevent fish from dying!
If you change to high O2 with this control,
this will directly changes O2 concentration in both aquariums.
Pretty neat!



Chapter 2: Shallow Copy

Both black boxes are controlled together!

```
# changing nested lists
my_aquarium_2[0][1] = "On"
print(my_sis_aquarium_2)
>>>[['50 % O2', 'On'], 'big fish', 'small fish',
'second big fish', 'Shark']
print(my_aquarium_2)
>>>[['50 % O2', 'On'], 'big fish', 'small fish',
'second big fish', 'Rainbow fish']
```

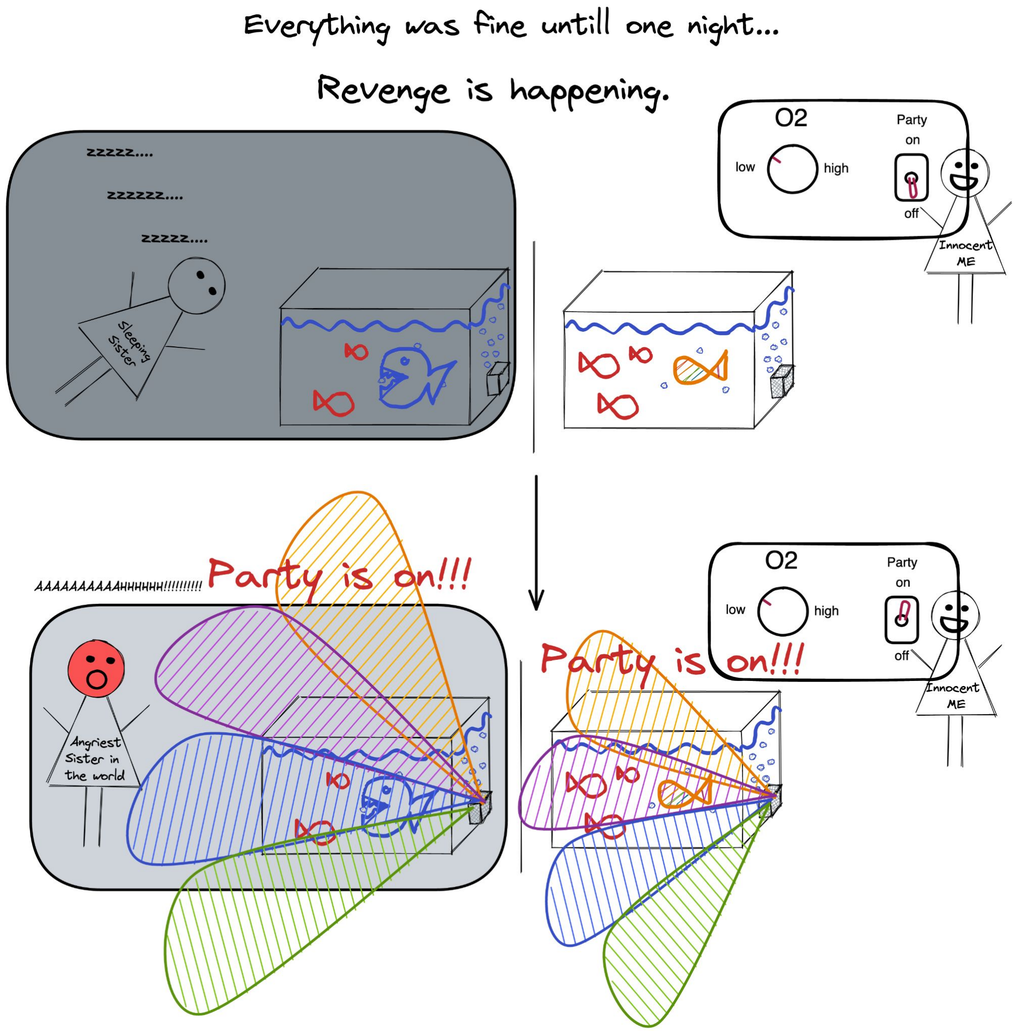


Chapter 2: Shallow Copy

Both black boxes are controlled together!

This is cool. Until… it isn't anymore.
Maybe my sister and me don't want to have a party at the same time…

So, my Sister in her rage, destroyed both aquariums!



Chapter 2: Shallow Copy -What happened?

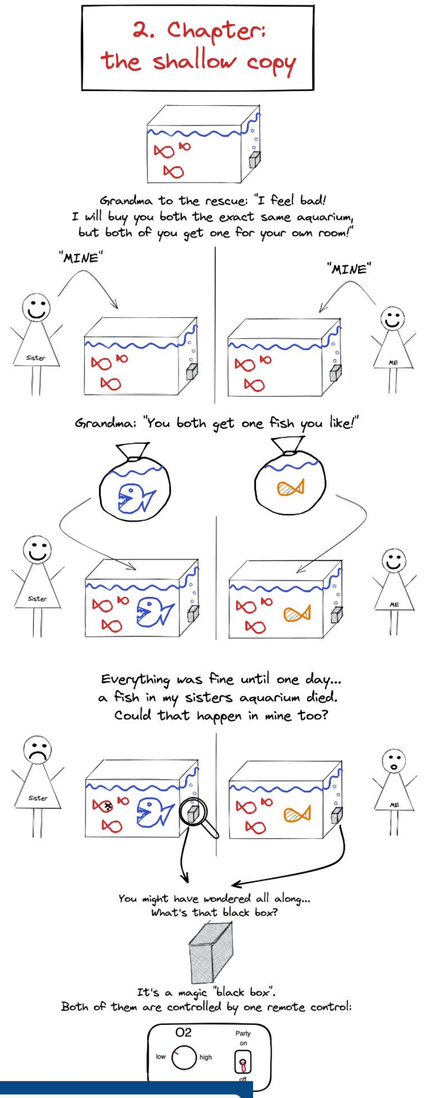
The shallow copy creates a new object of the list which stores the reference of the original elements.
Nested objects, the black box in our example (a list in a list), **aren't copied recursively.** Only the reference is copied!

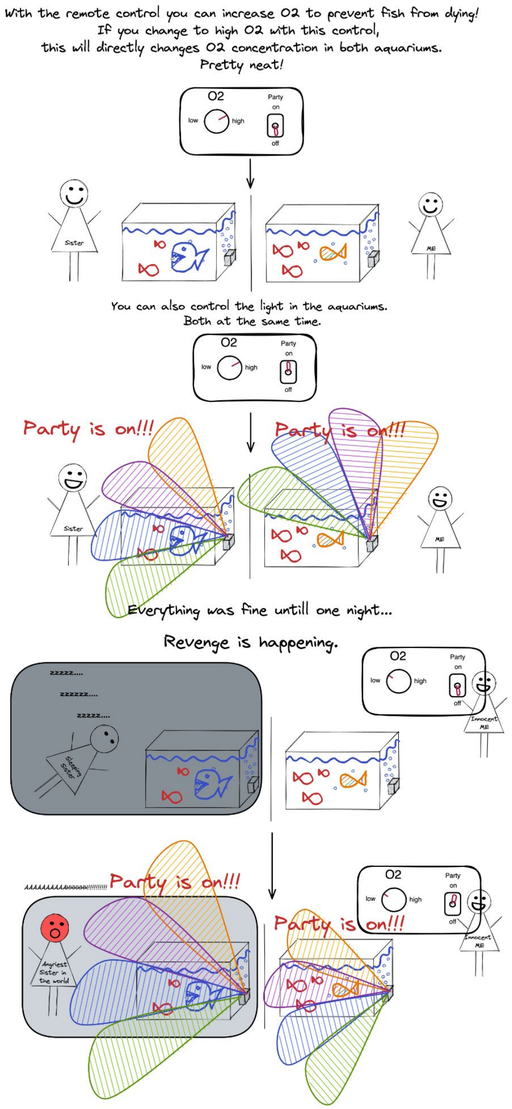
→ “my_aquarium_2” and “my_sis_aquarium_2” are different objects

```
# shallow copys don't share same id
id(my_aquarium_2) == id(my_sis_aquarium_2)
>>>False
```

→ all nested lists refer to the same object “blackbox”, changing one, will change all

```
# no copies are created of nested objects!
id(my_aquarium_2[0]) == id(my_sis_aquarium_2[0])
>>>True
id(my_aquarium_2[0]) == id(blackbox)
>>>True
```

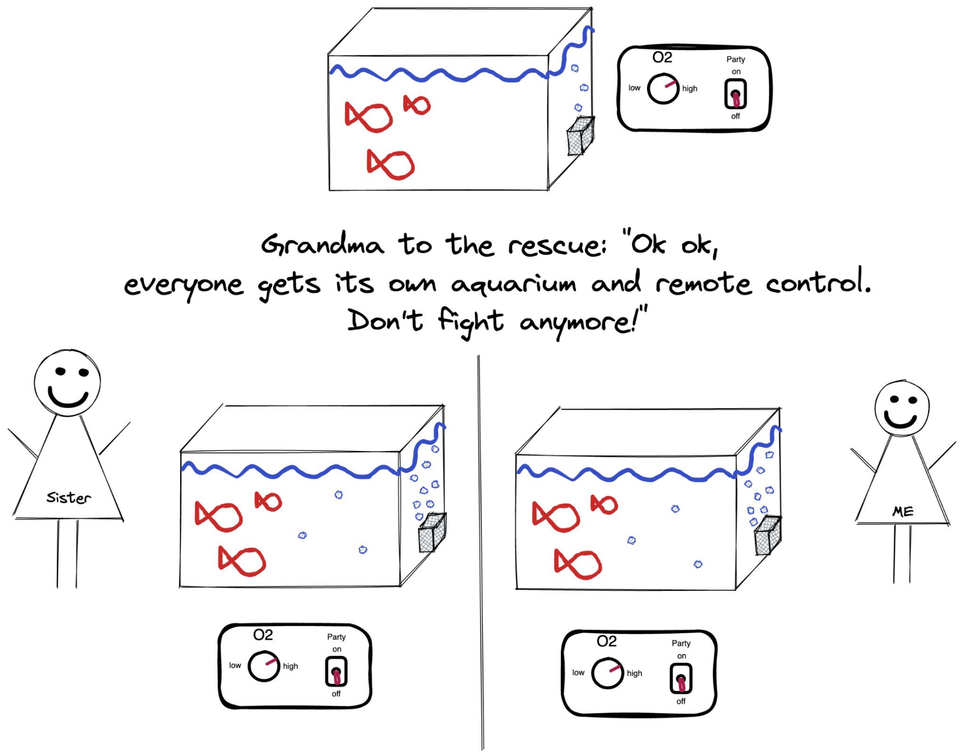




Chapter 3: Deep Copy

We are again at the start...

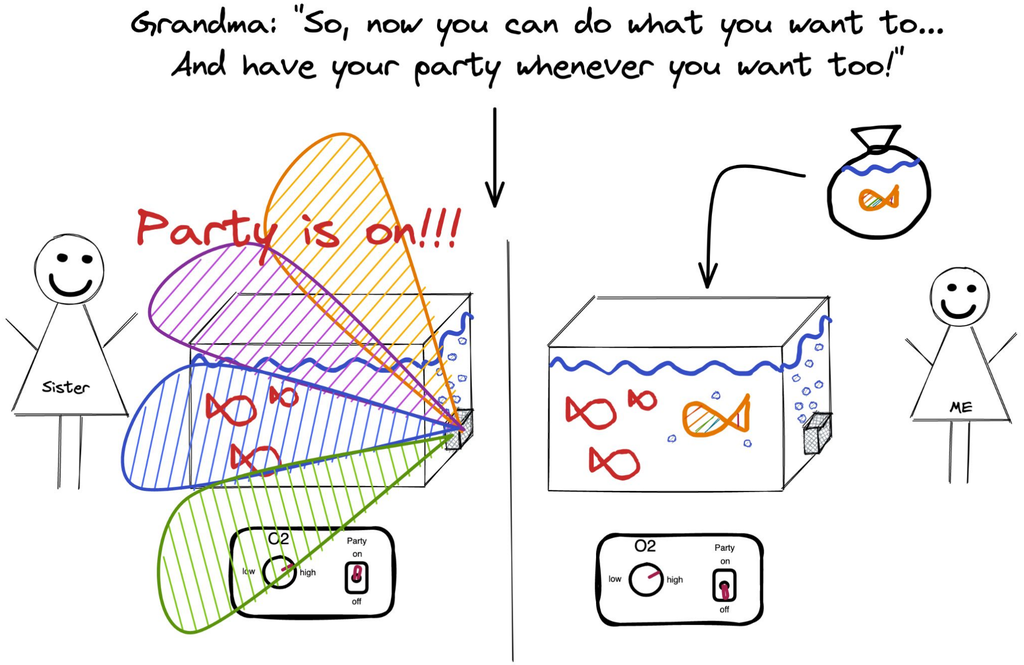
```
# deep copy
blackbox_2 = ['20 % O2', 'Off']
my_sis_aquarium_3 = [blackbox_2, 'big fish',
'small fish', 'second small fish']
my_aquarium_3 = copy.deepcopy(my_sis_aquarium_3)
```



Chapter 3: Deep Copy

Nothing I change in my aquarium will affect my sisters aquarium. And vice versa...

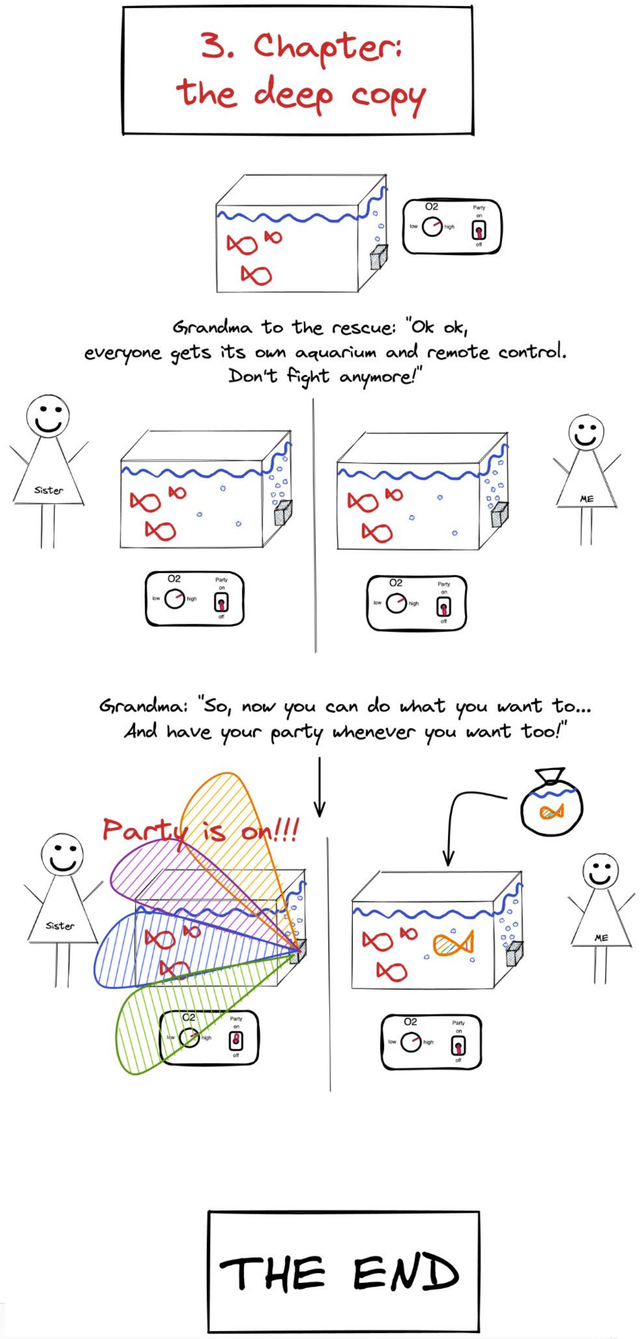
```
# nothing affects the other object
my_aquarium_3.append('Rainbow fish')
my_sis_aquarium_3[0][1] ="Party"
print(my_aquarium_3)
>>>[['20 % O2', 'Off'], 'big fish', 'small fish',
'second small fish', 'Rainbow fish']
print(my_sis_aquarium_3)
>>>[['20 % O2', 'Party'], 'big fish', 'small fish',
'second small fish']
```



Chapter 3: Deep Copy -What happened?

Object are copied and all nested objects are copied recursively as well!

```
# copies are created
id(my_aquarium_3) == id(my_sis_aquarium_3)
>>>False
# copies are created of nested objects!
id(my_aquarium_3[0]) == id(my_sis_aquarium_3[0])
>>>False
```



Summary:

- **“=” operator:** will not create a new object but will create a new reference to the old object!

→ don’t use “=” operator to copy mutable objects!

- **shallow copy:** will create a new object
  stores the reference of the original elements

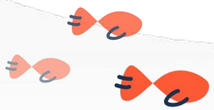
→ no copy of nested objects!

- **deep copy:** will create a new object with also copied nested objects!

→ totally independent!

When to use what?

- **“=” operator:**
  - \+ you can copy immutable objects
  - \- don’t copy mutable objects
- **shallow copy:**
  - \+ fast computation, use if you don’t have nested objects!
  - \+ if you want to have nested object (“one source of truth”) changing all copies in place
  - \- don’t use if you want copies of nested objects (totally independent copies)
- **deep copy:**
  - \+ totally independent of original
  - \- slow computation



QUIZZZZ:

That was all about lists…
how about dictionaries?

What's the result here?

A: True
B: False
C: The line above will cause an error as "=" operator not applicable for dicts
D: I don't know

```python
aquarium_dict = {
    'fishes': ['big fish', 'small fish', 'second big fish'],
    'plants': None,
    'black box': 1
}
my_aquarium_dict = my_sis_aquarium_dict = aquarium_dict

# whats the result here?
id(my_aquarium_dict) == id(my_sis_aquarium_dict)
```

QUIZZZ:

That was all about lists…
how about dictionaries?

What's the result here?

```python
aquarium_dict = {
    'fishes': ['big fish', 'small fish', 'second big fish'],
    'plants': None,
    'black box': 1
}
my_aquarium_dict = my_sis_aquarium_dict = aquarium_dict

# whats the result here?
id(my_aquarium_dict) == id(my_sis_aquarium_dict)
>>>True
```

A: True
B: False
C: The line above will cause an error as "=" operator not applicable for dicts
D: I don't know

QUIZZZZ:

That was all about lists…
how about dictionaries?

What's the result here?

```python
aquarium_dict = {
    'fishes': ['big fish', 'small fish', 'second big fish'],
    'plants': None,
    'black box': 1
}
my_aquarium_dict2 = aquarium_dict.copy()
my_sis_aquarium_dict2 = aquarium_dict.copy()

my_sis_aquarium_dict2['black box'] = "Party!!!"
# whats the result here?
id(my_aquarium_dict2['black box']) == id(my_sis_aquarium_dict2['black
box'])
```

A: True
B: False
C: The line above will cause an error
D: I don't know

QUIZZZZ:

That was all about lists…
how about dictionaries?

What's the result here?

```
aquarium_dict = {
    'fishes': ['big fish', 'small fish', 'second big fish'],
    'plants': None,
    'black box': 1
}
my_aquarium_dict2 = aquarium_dict.copy()
my_sis_aquarium_dict2 = aquarium_dict.copy()

my_sis_aquarium_dict2['black box'] = "Party!!!"
# whats the result here?
id(my_aquarium_dict2['black box']) == id(my_sis_aquarium_dict2['black
box'])
>>>False
```

A: True
B: False
C: The line above will cause an error
D: I don't know

QUIZZZZ:

That was all about lists…
how about dictionaries?

What's the result here?

```python
aquarium_dict = {
    'fishes': ['big fish', 'small fish', 'second big fish'],
    'plants': None,
    'black box': 1
}
my_aquarium_dict2 = aquarium_dict.copy()
my_sis_aquarium_dict2 = aquarium_dict.copy()

my_aquarium_dict2['fishes'].append("Rainbow fish")
# whats the result here?
id(my_aquarium_dict2['fishes']) == id(my_sis_aquarium_dict2['fishes'])
```

A: True
B: False
C: The line above will cause an Error.
D: I don't know

QUIZZZZ:

That was all about lists…
how about dictionaries?

What's the result here?

```python
aquarium_dict = {
    'fishes': ['big fish', 'small fish', 'second big fish'],
    'plants': None,
    'black box': 1
}
my_aquarium_dict2 = aquarium_dict.copy()
my_sis_aquarium_dict2 = aquarium_dict.copy()

my_aquarium_dict2['fishes'].append("Rainbow fish")
# whats the result here?
id(my_aquarium_dict2['fishes']) == id(my_sis_aquarium_dict2['fishes'])
>>>True
```

A: True
B: False
C: The line above will cause an error
D: I don't know

QUIZZZZ:

That was all about lists…
how about dictionaries?

What's the result here?

A: A shallow copy of aquarium_dict
B: A list of all dictionaries keys
C: The line above will cause an error
D: I don't know

```python
aquarium_dict = {
    'fishes': ['big fish', 'small fish', 'second big fish'],
    'plants': None,
    'black box': 1
}
my_aquarium_dict3 = list(aquarium_dict)
# whats the result here?
print(my_aquarium_dict3)
```

QUIZZZZ:

That was all about lists…
how about dictionaries?

What's the result here?

A: A shallow copy of aquarium_dict
B: A list of all dictionaries keys
C: The line above will cause an error
D: I don't know

→ for dictionaries you need the "dict" method to create shallow copies

```
aquarium_dict = {
    'fishes': ['big fish', 'small fish', 'second big fish'],
    'plants': None,
    'black box': 1
}
my_aquarium_dict3 = list(aquarium_dict)
# whats the result here?
print(my_aquarium_dict3)
>>>['fishes', 'plants', 'black box']
```

```
my_aquarium_dict3 = dict(aquarium_dict)
# whats the result here?
print(my_aquarium_dict3)
>>>{'fishes': ['big fish', 'small fish', 'second big fish'],
 'plants': None,
 'black box': 1}
```

Short break

How to use pythons generator objects

Or: The happy ending

Yay! Python 3!

... you all know lists.

... you all know several ways to create lists including list comprehensions.

... you all know how to copy them.

now let’s try to avoid making lists

11:30h.
On a casual Monday morning, you are idling over the second cappuccino when suddenly your boss is texting you.

Your annoying boss

Hello.
We need to talk.

Your eager self

Hi,
how can i help you?

Your annoying boss

...I need a list with all the prime numbers up to... 1,000,000 in python. No make it 10,000,000.

Answer 1:
List Comprehension

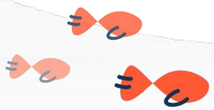
Your eager self

Sure. Just use a list comprehension like so:

```
start= datetime.now()

prime_list=[i for i in range(10000000) if is_prime(i)]

duration =round((datetime.now()-start).total_seconds(),2)

p_say(f"Done. result: ...{prime_list[-5:]}")
p_say(f"Computation took {duration} s")
```

```
Python:  Done. result: ...[9999937, 9999943, 9999971, 9999973, 9999991]
Python:  Computation took 62.3 s
```

About an hour later



Your annoyed boss

Thanks, that works. But It’s too slow. You need to do better.

Your lazy self

Well then I need more information. What you need the list for anyways?

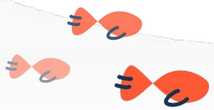
Your annoyed boss

~~None of your Business~~

…. I want to look through that list to find prime siblings! I reckon I should be able to find 5,000 in that list. Or do I need more primes?

Your lazy self

That sounds ~~stupid~~ interesting. Let’s just build you a generator for prime numbers instead. I’ll get right to it.



Answer 2:
Generator

Done, here:

Your lazy self

```
start =datetime.now()

prime_list_gen=(i for i in range(10000000) if is_prime(i))

duration =round((datetime.now()-start).total_seconds(),2)

p_say(f"Done. result: {prime_list_gen}")
p_say(f"Computation took {duration} s")
```

```
Python:  Done. result: <generator object <genexpr> at 0x118243970>
Python:  Computation took 0.0 s
```

To get the next prime number, just call

```
next(prime_list_gen)
```

Let’s first compare the two answers

List comprehension

```
start= datetime.now()

prime_list=[i for i in range(10000000) if is_prime(i)]

duration =round((datetime.now()-start).total_seconds(),2)

p_say(f"Done. result: ...{prime_list[-5:]}")
p_say(f"Computation took {duration} s")
```

```
Python:  Done. result: ...[9999937, 9999943, 9999971, 9999973, 9999991]
Python:  Computation took 62.3 s
```

round vs squared brackets

Generator

```
start =datetime.now()

prime_list_gen=(i for i in range(10000000) if is_prime(i))

duration =round((datetime.now()-start).total_seconds(),2)

p_say(f"Done. result: {prime_list_gen}")
p_say(f"Computation took {duration} s")
```

```
Python:  Done. result: <generator object <genexpr> at 0x118243970>
Python:  Computation took 0.0 s
```

Let’s first compare the two answers

List comprehension

```
start= datetime.now()

prime_list=[i for i in range(10000000) if is_prime(i)]

duration =round((datetime.now()-start).total_seconds(),2)

p_say(f"Done. result: ...{prime_list[-5:]}")
p_say(f"Computation took {duration} s")
```

```
Python:  Done. result: ...[9999937, 9999943, 9999971, 9999973, 9999991]
Python:  Computation took 62.3 s
```

Python!
Here is a recipe.

I need a list. Now!

Eager evaluation

Generator

```
start =datetime.now()

prime_list_gen=(i for i in range(10000000) if is_prime(i))

duration =round((datetime.now()-start).total_seconds(),2)

p_say(f"Done. result: {prime_list_gen}")
p_say(f"Computation took {duration} s")
```

```
Python:  Done. result: <generator object <genexpr> at 0x118243970>
Python:  Computation took 0.0 s
```

Python!
Here is a recipe.

I need a list. … But, let me get back to you when I actually need it ok?

Lazy evaluation

Let’s first compare the two answers

List comprehension

```
start= datetime.now()

prime_list=[i for i in range(10000000) if is_prime(i)]

duration =round((datetime.now()-start).total_seconds(),2)

p_say(f"Done. result: ...{prime_list[-5:]}")
p_say(f"Computation took {duration} s")
```

```
Python:  Done. result: ...[9999937, 9999943, 9999971, 9999973, 9999991]
Python:  Computation took 62.3 s
```

- Works fine!
- Slower (careful with upper limit)
- Uses all the memory
- Afterwards you can be sure the preparation is done and successful

Generator

```
start =datetime.now()

prime_list_gen=(i for i in range(10000000) if is_prime(i))

duration =round((datetime.now()-start).total_seconds(),2)

p_say(f"Done. result: {prime_list_gen}")
p_say(f"Computation took {duration} s")
```

```
Python:  Done. result: <generator object <genexpr> at 0x118243970>
Python:  Computation took 0.0 s
```

- Don’t need to be careful with upper limit
- Better breakpoint in case of problems
- Don’t clog up memory if not needed

Let’s first compare how list comprehensions and generators are defined

List comprehension

```
prime_list=[i for i in range(1000000) if is_prime(i)]
```

“[“: 3 strokes: eager

round vs squared brackets

“(“ 1 stroke: lazy

Generator

```
prime_list_gen=(i for i in range(1000000) if is_prime(i))
```

But how do we get results from the generator?

```
start =datetime.now()

prime_list_gen=(i for i in range(10000000) if is_prime(i))

duration =round((datetime.now()-start).total_seconds(),2)

p_say(f"Done. result: {prime_list_gen}")
p_say(f"Computation took {duration} s")
```

```
Python:  Done. result: <generator object <genexpr> at 0x118243970>
Python:  Computation took 0.0 s
```

Easy!

```
next(prime_list_gen)
```

```
2
```

```
[next(prime_list_gen) for _ in range(15)]
```

```
[3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53]
```

```
prime_list=[i for i in prime_list_gen]

print(f"{prime_list[:5]} ...  {prime_list[-5:]}")
```

```
[59, 61, 67, 71, 73] ...  [999953, 999959, 999961, 999979, 999983]
```

Alternative syntax for generator

function

```
def complicatedFunction():
    #Do some complicated stuff here
    return something
```

generator

```
def complicatedGenerator():
    #Do some complicated stuff here
    yield something
```

yield vs return

Alternative syntax for generator

From function....

```
def is_prime(number):
    if number <2: return False
    if number in [2,3,5,7]: return True

    for i in range(2,int(number**0.5)+1):
        if number%i==0:
            return False
    return True
```

To generator

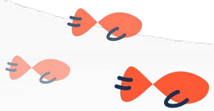
```
def prime_generator():
    yield 2 #manually yield 2 first, so that we can start with 3 and use an increment by 2
    number=3
    while True:
        is_Prime=True
        for i in range(2,int(number**0.5)+1):
            if number%i==0:
                is_Prime=False
                break
        if is_Prime: yield number
        number+=2

prime_gen=prime_generator()
```

```
[next(prime_list_gen) for _ in range(10)]
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Fundamentals 1: List comprehension

So let’s visit the notebook!!



Thank you for your attention!