CS 61A DISCUSSION 4

LIST MUTATION, ORDERS OF GROWTH, AND NONLOCAL

Raymond Chan
Discussion 134
UC Berkeley Fall 16

AGENDA

- Announcements
- List Mutation
- Orders of Growth
- Nonlocal
- Appendix
  - Dictionaries

ANNOUNCEMENTS

- Maps due tonight
- Lab 5 due Friday 9/30
- CSM signups reopened
- 61A one-on-one tutoring

CHALLENGE QUESTION

- For those who runs through the packet, what is the order of the growth for the function below?

```
def f(n):
  i = 2
  while i < n:
    print(i)
    i = i * i
```

CHALLENGE QUESTION

- For those who runs through the packet, what is the order of the growth for the function below?

```
def f(n):
  i = 2
  while i < n:
    print(i)
    i = i * i
```

$\Theta(\log \log n)$

CHALLENGE QUESTION

- For those who runs through the packet, what is the order of the growth for the function below?

```
def f(n):
  i = 2
  while i < n:
    print(i)
    i = i * i
```

Θ(log log n)

32
31
.
.
16
15
.
.
3
2
1

Θ(1)

32
16
8
4
2
1

Θ(log n)

32
4
2
1

Θ(log log n)

MUTATION

- When we define functions, we created function objects in environment diagrams.
- When we create lists, we create list objects.
- We can change the elements of list objects after we've created it.

```
>>> a =[1, 2, 3]
>>> a
[1, 2, 3]
>>> a[2] = 100
>>> a
```

MUTATION

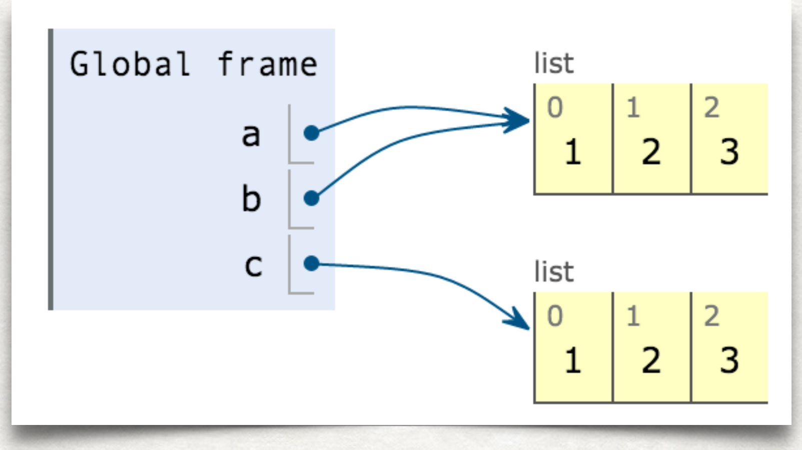
- When we define functions, we created function objects in environment diagrams.
- When we create lists, we create list objects.
- We can change the elements of list objects after we've created it.

```
>>> a =[1, 2, 3]
>>> a
[1, 2, 3]
>>> a[2] = 100
>>> a
[1, 2, 100]
```

MUTATION

- If I assign this variable **a** to variable **b**, **b** receives the reference.
- **a** and **b** is the same list as they are both referencing the same list object
- **a**, **b**, and **c** have the same elements, but a and c are not the same list

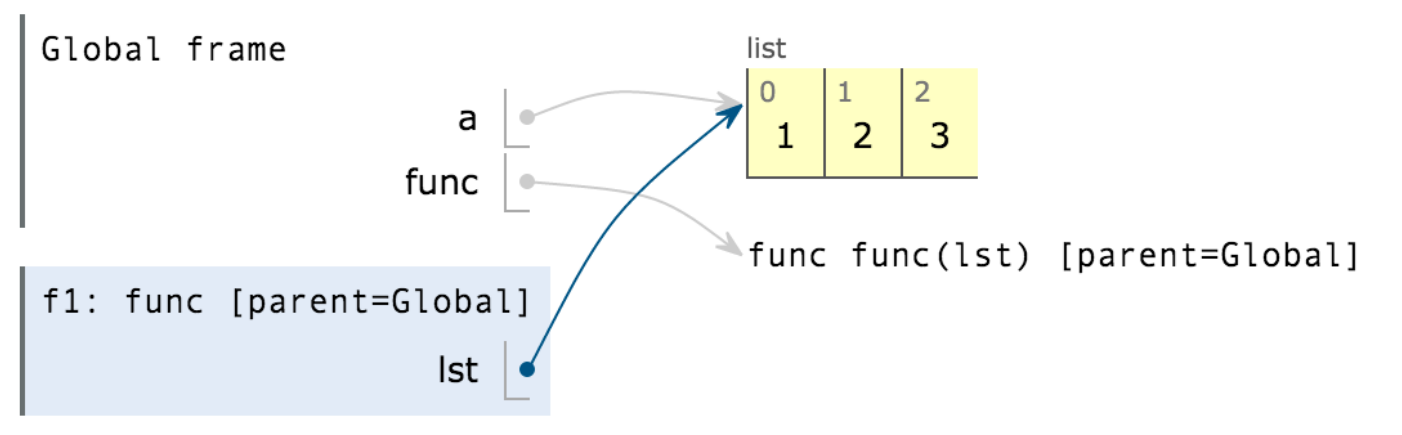
```
a = [1, 2, 3]
b = a
c = [1,2, 3]
```



MUTATION

- When we assign a list to a variable, the variable references the list object.
- If I pass in a variable that references a list to a function argument, I am passing in the reference.
    - This is similar to passing in a function object.

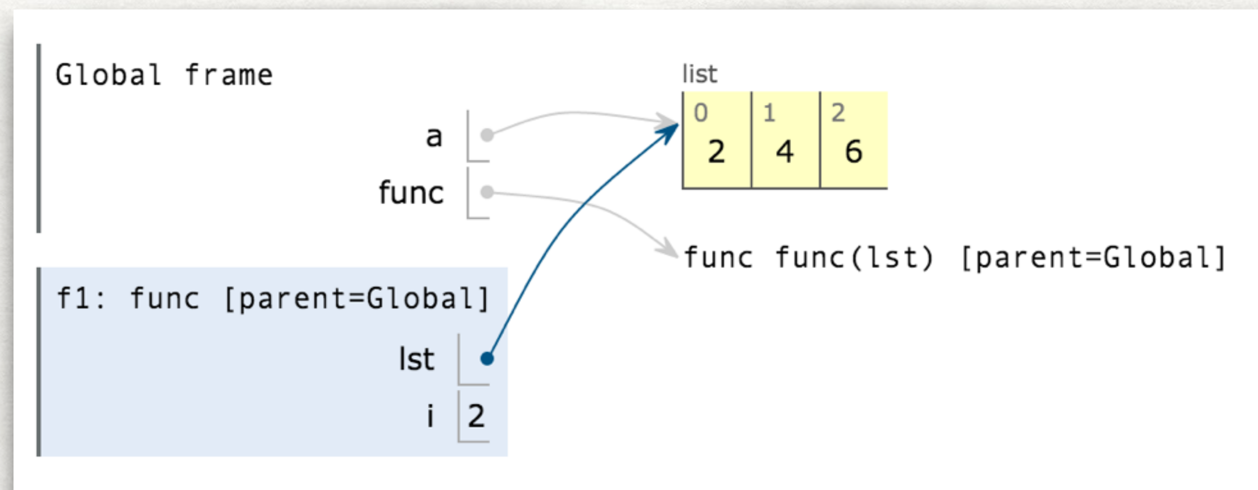
```
a = [1, 2, 3]
def func(lst):
    for i in range(0, len(lst)):
        lst[i] = lst[i] * 2

func(a)
```



MUTATION

- Within the body of **func**, lst's values are changed. Notice that a's values are also changed because lst references the same list a is point to.

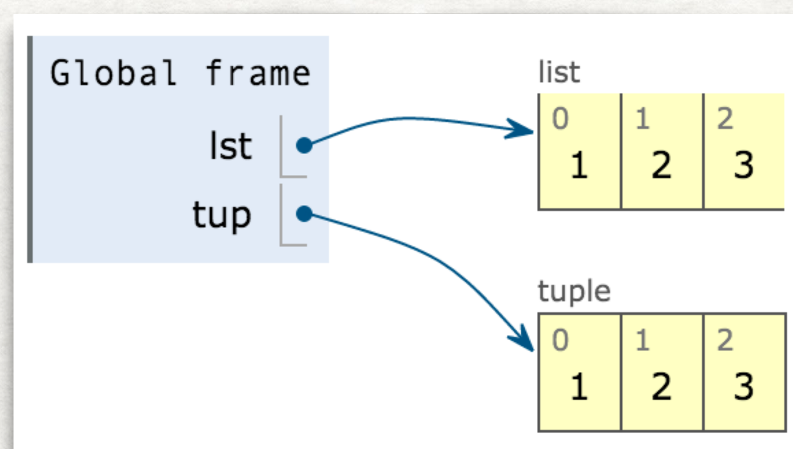
```
a = [1, 2, 3]
def func(lst):
    for i in range(0, len(lst)):
        lst[i] = lst[i] * 2

func(a)
```



MUTATION

- Lists and dictionaries are mutable.
- Tuples and strings are immutable. Once they are created, they cannot be changed.

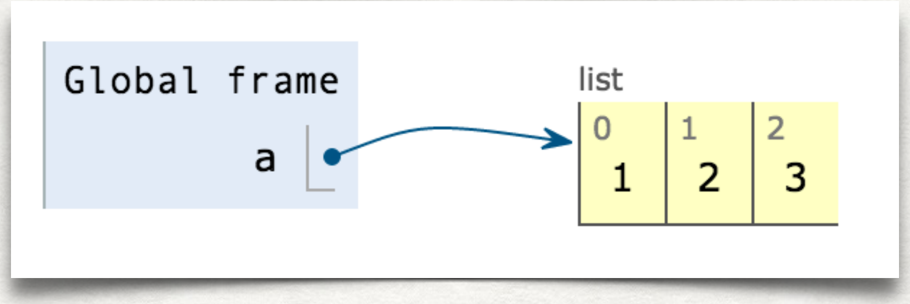
```
lst = [1, 2, 3]
tup = (1, 2, 3)
```

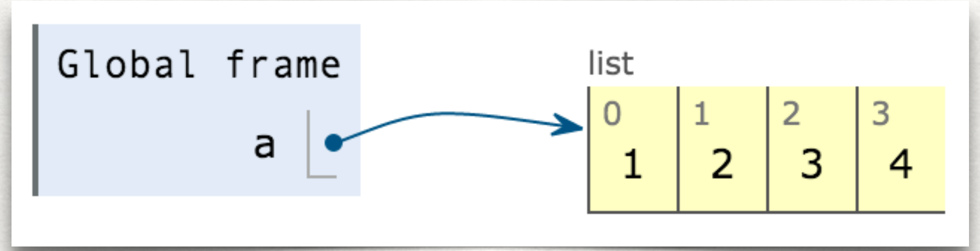


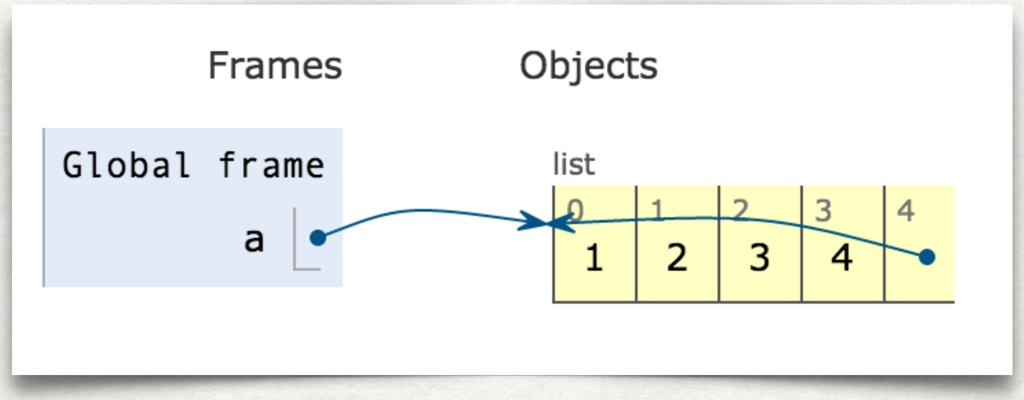
MUTATION

- **lst.append(x)** adds x to the end of the list.
- Only creates **one** new index.

```
>>> a = [1, 2, 3]
>>> a.append(4)
>>> a
[1, 2, 3, 4]
>>> a.append([5, 6])
>>> a
[1, 2, 3, 4, [5, 6]]
>>> len(a)
5
```



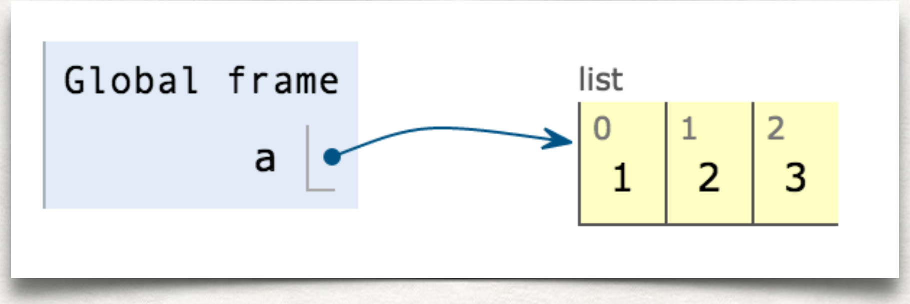


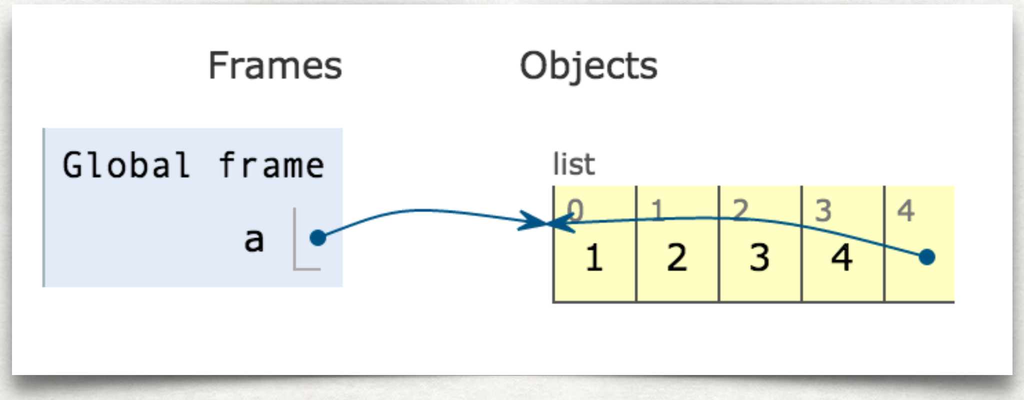


MUTATION

- A list can append itself.

```
>>> a = [1, 2, 3, 4]
>>> a.append(a)
>>> a
[1, 2, 3, 4, [...]]
>>> a[4][3]
4
>>> a[4][4][4][2]
3
```

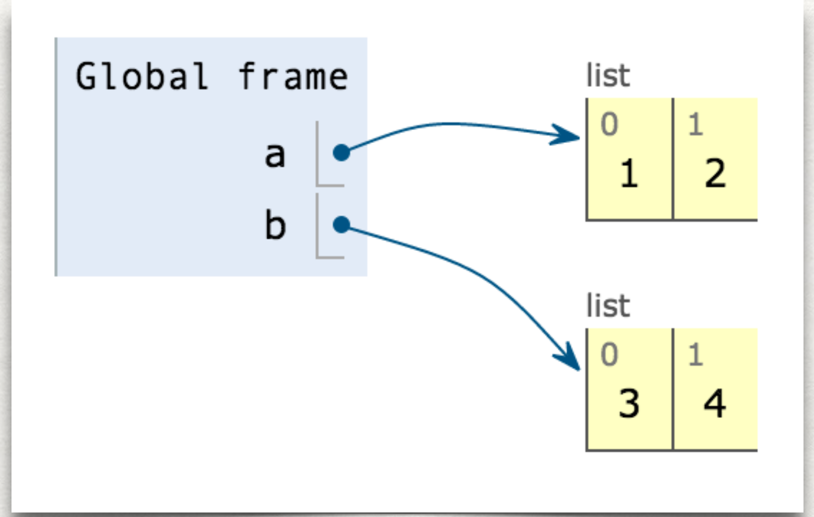


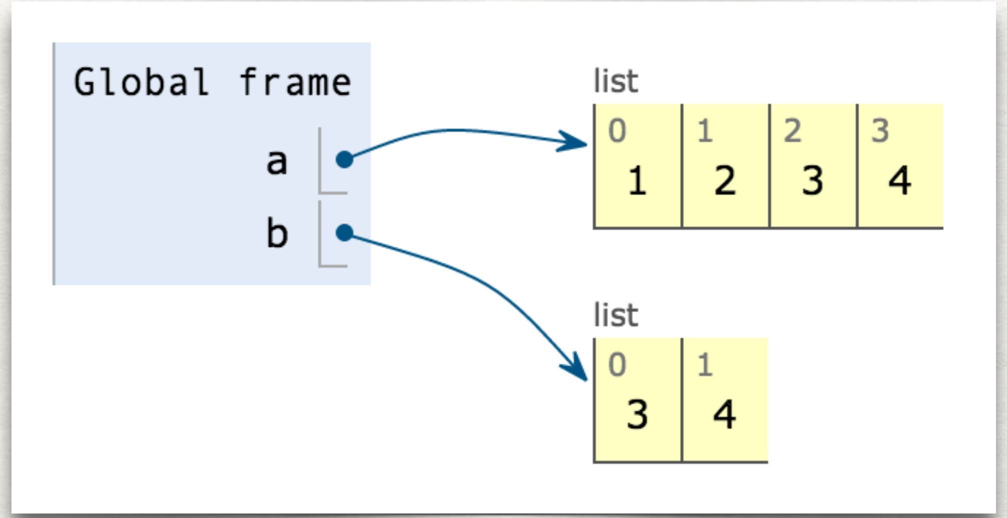


MUTATION

- **lst.extend(seq)** appends each element of seq to list.
- **seq** can be a list or a tuple.
- tinyurl.com/mutation-q1

```
>>> a = [1, 2]
>>> b = [3, 4]
>>> a.extend(b)
>>> a
[1, 2, 3, 4]
>>> b
[3, 4]
```

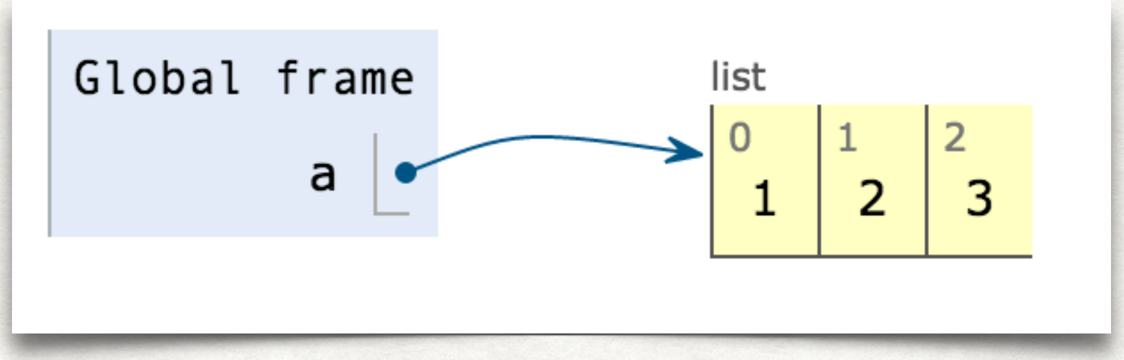


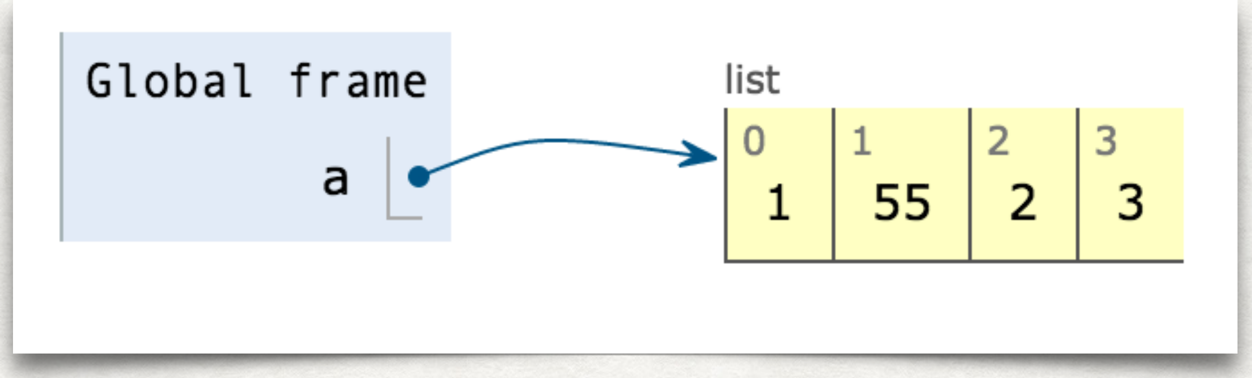


MUTATION

- **lst.insert(i, x)** inserts **x** at index **i** by adding a new element and not replace the original element at **i**.

```
>>> a = [1, 2, 3]
>>> a.insert(1, 55)
>>> a
[1, 55, 2, 3]
```

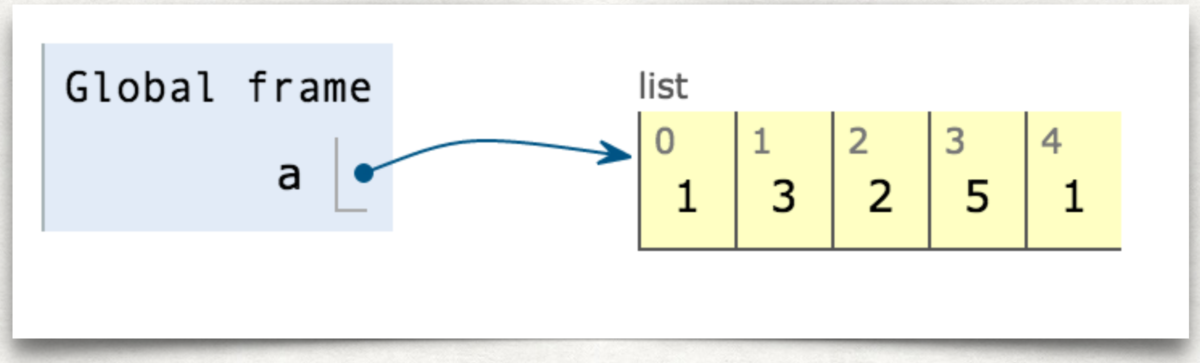


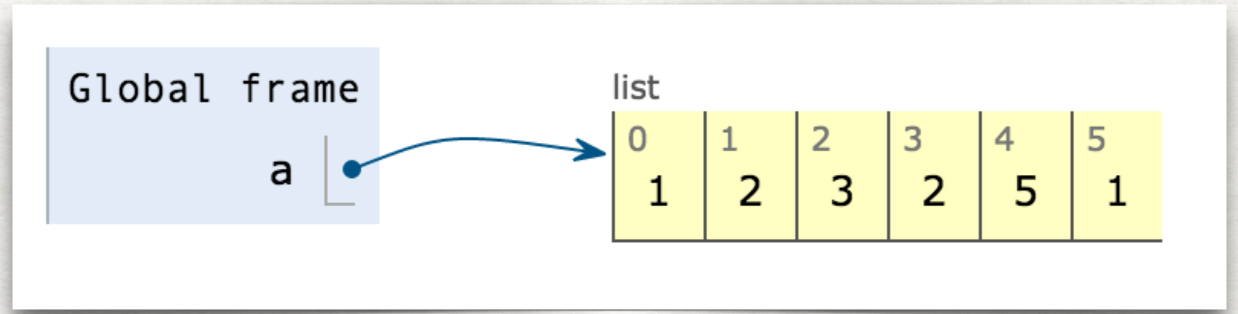


MUTATION

- **lst.remove(x)** removes the first time we see x in a list, otherwise errors

```
>>> a = [1, 2, 3, 2, 5, 1]
>>> a.remove(2)
>>> a
[1, 3, 2, 5, 1]
```

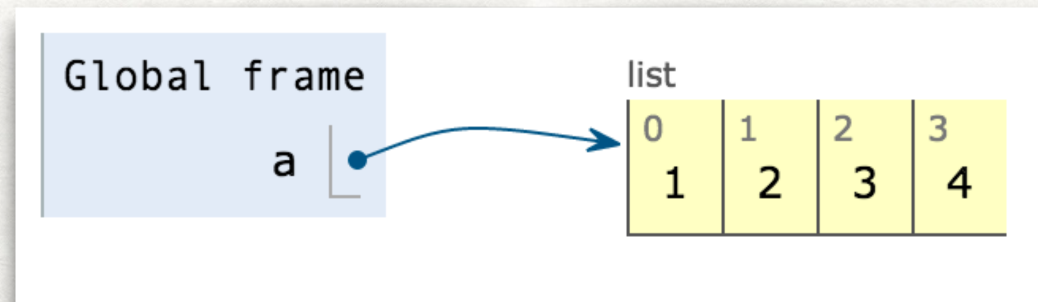


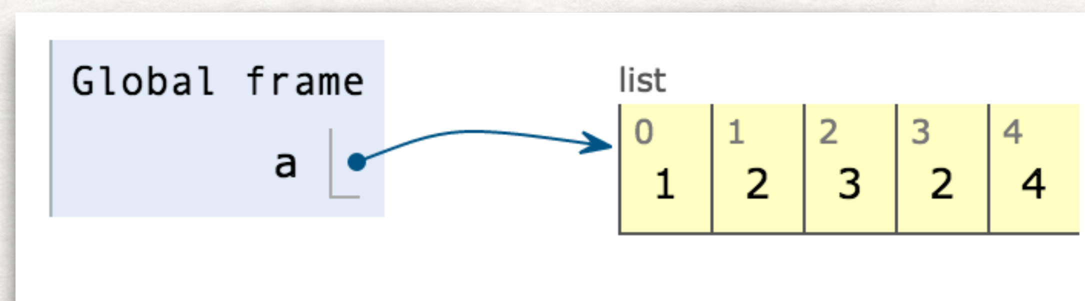


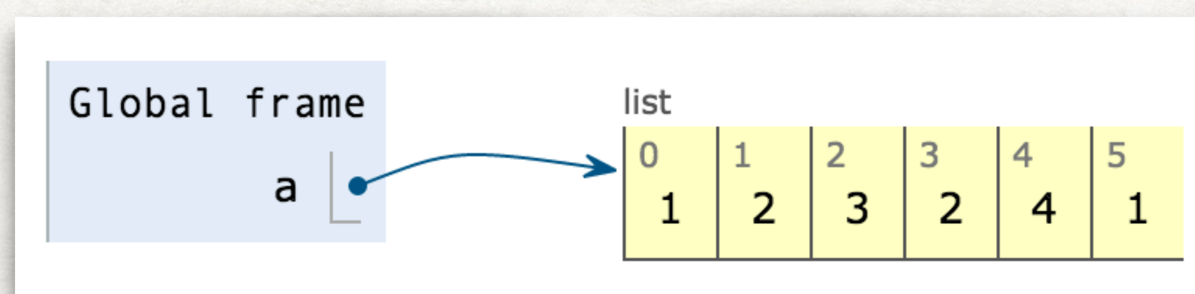
MUTATION

- **lst.pop(i)** removes and returns the element at index i. By default, i is the last element.

```
>>> a = [1, 2, 3, 2, 4, 1]
>>> a.pop()
1
>>> a.pop(3)
2
>>> a
[1, 2, 3, 4]
```







MUTATION

- **+=** for lists mutates the original list.
- **+=** is different from **a = a + [1]** when **a is a <u>list.</u>**
- Evaluating right hand side creates a **new** list and then assigns the nest list to **a.**

```
>>> a = [1, 2, 3, 4]
>>> id(a)
<some id 1>
>>> a += [3]
>>> a
[1, 2, 3, 4, 3]
>>> id(a)
<some id 1>
```

```
>>> a = a + [2]
>>> a
[1, 2, 3, 4, 3, 2]
>>> id(a)
>>>
<some id 2>
```

MUTATION

- **+=** for lists mutates the original list, but is still a "reassignment".
- Thus the list needs to be in the local frame.
- Using **append** or **extend** only require access to the list.
- It can be in the parent frame.

MUTATION

```
lst = [1, 2, 3]
def f():
   lst.append(4)

f()
print(lst)
```

```
def g():
   lst += [5]

g()
```

```
[1, 2, 3, 4]
```

```
Error
```

MUTATION

```
>>> lst1 = [1, 2, 3]
>>> lst2 = [1, 2, 3]
#compares each value
>>> lst1 == lst2

#compares references
>>> lst1 is lst2

>>> lst2 = lst1
>>> lst2 is lst1

>>> lst1.append(4)
>>> lst1

>>> lst2
```

```
>>> lst2[1] =42
>>> lst2

>>> lst1 = lst1 + [5]
>>> lst1 == lst2

>>> lst1

>>> lst2

>>> lst2 is lst1
```

MUTATION

```
>>> lst1 = [1, 2, 3]
>>> lst2 = [1, 2, 3]
#compares each value
>>> lst1 == lst2
True
#compares references
>>> lst1 is lst2

>>> lst2 = lst1
>>> lst2 is lst1

>>> lst1.append(4)
>>> lst1

>>> lst2
```

```
>>> lst2[1] =42
>>> lst2

>>> lst1 = lst1 + [5]
>>> lst1 == lst2

>>> lst1

>>> lst2

>>> lst2 is lst1
```

MUTATION

```
>>> lst1 = [1, 2, 3]
>>> lst2 = [1, 2, 3]
#compares each value
>>> lst1 == lst2
True
#compares references
>>> lst1 is lst2
False
>>> lst2 = lst1
>>> lst2 is lst1

>>> lst1.append(4)
>>> lst1

>>> lst2
```

```
>>> lst2[1] =42
>>> lst2

>>> lst1 = lst1 + [5]
>>> lst1 == lst2

>>> lst1

>>> lst2

>>> lst2 is lst1
```

MUTATION

```
>>> lst1 = [1, 2, 3]
>>> lst2 = [1, 2, 3]
#compares each value
>>> lst1 == lst2
True
#compares references
>>> lst1 is lst2
False
>>> lst2 = lst1
>>> lst2 is lst1
True
>>> lst1.append(4)
>>> lst1

>>> lst2
```

```
>>> lst2[1] =42
>>> lst2

>>> lst1 = lst1 + [5]
>>> lst1 == lst2

>>> lst1

>>> lst2

>>> lst2 is lst1
```

MUTATION

```
>>> lst1 = [1, 2, 3]
>>> lst2 = [1, 2, 3]
#compares each value
>>> lst1 == lst2
True
#compares references
>>> lst1 is lst2
False
>>> lst2 = lst1
>>> lst2 is lst1
True
>>> lst1.append(4)
>>> lst1
[1, 2, 3, 4]
>>> lst2
```

```
>>> lst2[1] =42
>>> lst2

>>> lst1 = lst1 + [5]
>>> lst1 == lst2

>>> lst1

>>> lst2

>>> lst2 is lst1
```

MUTATION

```
>>> lst1 = [1, 2, 3]
>>> lst2 = [1, 2, 3]
#compares each value
>>> lst1 == lst2
True
#compares references
>>> lst1 is lst2
False
>>> lst2 = lst1
>>> lst2 is lst1
True
>>> lst1.append(4)
>>> lst1
[1, 2, 3, 4]
>>> lst2
[1, 2, 3, 4]
```

```
>>> lst2[1] =42
>>> lst2

>>> lst1 = lst1 + [5]
>>> lst1 == lst2

>>> lst1

>>> lst2

>>> lst2 is lst1
```

MUTATION

```
>>> lst1 = [1, 2, 3]
>>> lst2 = [1, 2, 3]
#compares each value
>>> lst1 == lst2
True
#compares references
>>> lst1 is lst2
False
>>> lst2 = lst1
>>> lst2 is lst1
True
>>> lst1.append(4)
>>> lst1
[1, 2, 3, 4]
>>> lst2
[1, 2, 3, 4]
```

```
>>> lst2[1] =42
>>> lst2
[1, 42, 3, 4]
>>> lst1 = lst1 + [5]
>>> lst1 == lst2

>>> lst1

>>> lst2

>>> lst2 is lst1
```

MUTATION

```
>>> lst1 = [1, 2, 3]
>>> lst2 = [1, 2, 3]
#compares each value
>>> lst1 == lst2
True
#compares references
>>> lst1 is lst2
False
>>> lst2 = lst1
>>> lst2 is lst1
True
>>> lst1.append(4)
>>> lst1
[1, 2, 3, 4]
>>> lst2
[1, 2, 3, 4]
```

```
>>> lst2[1] =42
>>> lst2
[1, 42, 3, 4]
>>> lst1 = lst1 + [5]
>>> lst1 == lst2
False
>>> lst1

>>> lst2

>>> lst2 is lst1
```

MUTATION

```
>>> lst1 = [1, 2, 3]
>>> lst2 = [1, 2, 3]
#compares each value
>>> lst1 == lst2
True
#compares references
>>> lst1 is lst2
False
>>> lst2 = lst1
>>> lst2 is lst1
True
>>> lst1.append(4)
>>> lst1
[1, 2, 3, 4]
>>> lst2
[1, 2, 3, 4]
```

```
>>> lst2[1] =42
>>> lst2
[1, 42, 3, 4]
>>> lst1 = lst1 + [5]
>>> lst1 == lst2
False
>>> lst1
[1, 42, 3, 4, 5]
>>> lst2

>>> lst2 is lst1
```

MUTATION

```
>>> lst1 = [1, 2, 3]
>>> lst2 = [1, 2, 3]
#compares each value
>>> lst1 == lst2
True
#compares references
>>> lst1 is lst2
False
>>> lst2 = lst1
>>> lst2 is lst1
True
>>> lst1.append(4)
>>> lst1
[1, 2, 3, 4]
>>> lst2
[1, 2, 3, 4]
```

```
>>> lst2[1] =42
>>> lst2
[1, 42, 3, 4]
>>> lst1 = lst1 + [5]
>>> lst1 == lst2
False
>>> lst1
[1, 42, 3, 4, 5]
>>> lst2
[1, 42, 3, 4]
>>> lst2 is lst1
```

MUTATION

```
>>> lst1 = [1, 2, 3]
>>> lst2 = [1, 2, 3]
#compares each value
>>> lst1 == lst2
True
#compares references
>>> lst1 is lst2
False
>>> lst2 = lst1
>>> lst2 is lst1
True
>>> lst1.append(4)
>>> lst1
[1, 2, 3, 4]
>>> lst2
[1, 2, 3, 4]
```

```
>>> lst2[1] =42
>>> lst2
[1, 42, 3, 4]
>>> lst1 = lst1 + [5]
>>> lst1 == lst2
False
>>> lst1
[1, 42, 3, 4, 5]
>>> lst2
[1, 42, 3, 4]
>>> lst2 is lst1
False
```

MUTATION

Write a function that removes all instances of an element from a list.

```
def remove_all(el, lst):
    """
    >>> x = [3, 1, 2, 1, 5, 1, 1, 7]
    >>> remove_all(1, x)
    >>> x
    [3, 2, 5, 7]
    """
```

MUTATION

Write a function that removes all instances of an element from a list.

```
def remove_all(el, lst):
    """
    >>> x = [3, 1, 2, 1, 5, 1, 1, 7]
    >>> remove_all(1, x)
    >>> x
    [3, 2, 5, 7]
    """
    while el in lst:
        lst.remove(el)
```

MUTATION

Write a function that takes two values **x1** and **el**, and a list, and adds as many **el**'s to the end of this lists there are **x**'s.

```
def add_this_many(x, el, lst):
    """ Adds el to the end of lst the number of times x occurs
    in lst.
    >>> lst = [1, 2, 4, 2, 1]
    >>> add_this_many(1, 5, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5]
    >>> add_this_many(2, 2, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5, 2, 2]
    """
```

MUTATION

Write a function that takes two values **x1** and **el**, and a list, and adds as many **el**'s to the end of this lists there are **x**'s.

```
def add_this_many(x, el, lst):
    """ Adds el to the end of lst the number of times x occurs
    in lst.
    >>> lst = [1, 2, 4, 2, 1]
    >>> add_this_many(1, 5, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5]
    >>> add_this_many(2, 2, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5, 2, 2]
    """
    count = 0
    for element in lst:
        if element == x:
            count += 1
    while count > 0:
        lst.append(el)
        count -= 1
```

MUTATION

Write a function that takes two values **x1** and **el**, and a list, and adds as many **el**'s to the end of this lists there are **x**'s.

```
def add_this_many(x, el, lst):
    """ Adds el to the end of lst the number of times x occurs
    in lst.
    >>> lst = [1, 2, 4, 2, 1]
    >>> add_this_many(1, 5, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5]
    >>> add_this_many(2, 2, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5, 2, 2]
    """
    count = 0
    for element in lst:
        if element == x:
            count += 1
    while count > 0:
        lst.append(el)
        count -= 1
```

```
for el in lst:
    if el == x:
        lst.append(el)
```

Wrong solutions because the elements are added to the end of the list as you iterate. Thus it could be iterating for ever.
add_this_many(2, 2, lst)

ORDERS OF GROWTH

- When we have really large inputs, we need to worry about efficiency.
- We measure efficiency by runtime (Time complexity).
- How long does the functions take to run in terms of the size of the input?
- If the size of the input grows, how does the runtime change?

ORDERS OF GROWTH

- We use Big-**Θ** notation means a tight bound on time complexity.
- **$\Theta(n^2)$** means that the function's runtime is no larger and no smaller than quadratic of the input.

ORDERS OF GROWTH

- ***n:*** size of problem
- ***R(n)***: amount of resource used (time or space)
- $R(n) = \Theta(f(n))$
- $k_1 * f(n) \leq R(n) \leq k_2 * f(n)$
- where k_1 and k_2 are some constants and $k_1 \leq k_2$
- Assume ***n*** is larger than some minimum ***m***

ORDERS OF GROWTH

```
def square(n):
   return n * n
```

1 primitive operation *

For our purposes, * is constant time

input	function call	number of	number of operations
1	square(1)	1*1	1
2	square(2)	2*2	1
…	…	…	…
100	square(100)	100*100	1
…	…	…	…
n	square(n)	n*n	1

ORDERS OF GROWTH

```
def square(n):
   return n * n
```

1 primitive operation *
For our purposes, * is constant time

Θ(1)

input	function call	number of	number of operations
1	square(1)	1*1	1
2	square(2)	2*2	1
…	…	…	…
100	square(100)	100*100	1
…	…	…	…
n	square(n)	n*n	1

ORDERS OF GROWTH

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Each recursive call has
a constant amount operations.
But we have **n** recursive calls

input	function call	return value	number of operations
1	factorial(1)	1*1	1
2	factorial(2)	2*1*1	2
…	…	…	…
100	factorial(100)	100*99*…*1*1	100
…	…	…	…
n	factorial(n)	n*(n-1)*…*1*1	n

ORDERS OF GROWTH

```
def factorial(n):
   if n == 0:
      return 1
   return n * factorial(n - 1)
```

Each recursive call has
a constant amount operations.
But we have **n** recursive calls

Θ(n)

input	function call	return value	number of operations
1	factorial(1)	1*1	1
2	factorial(2)	2*1*1	2
…	…	…	…
100	factorial(100)	100*99*…*1*1	100
…	…	…	…
n	factorial(n)	n*(n-1)*…*1*1	n

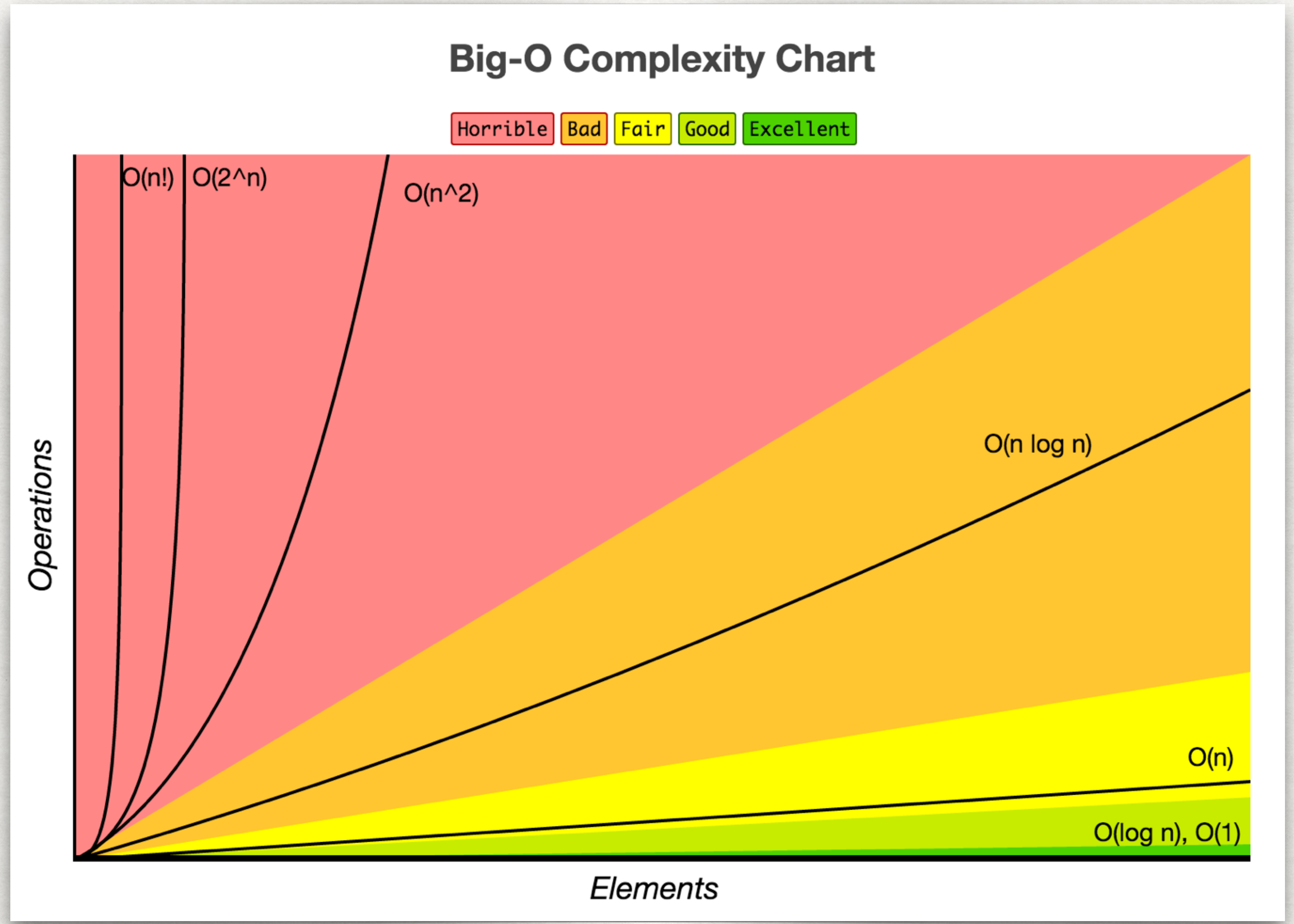
ORDERS OF GROWTH

- **ϴ(1)** - constant time; same time regardless of input size.
- **ϴ(log n)** - logarithmic time; e.g. usually dividing the problem down by some factor.
- **ϴ(n)** - linear time; e.g. usually 1 loop
- **ϴ(n^2)**, **ϴ(n^3)**, etc - polynomial time; e.g. nested loops
- **ϴ(c^n)** - exponential time; where **c** is some constant; really horrible time complexity; e.g. tree recursion

ORDERS OF GROWTH

- Constant time is the best and exponential is the worse.
- Any polynomial is worse than any logarithmic.
- Higher degree polynomial worse than lower degree.

ORDERS OF GROWTH



For this class, assume O is **Θ**.

See appendix, for other runtime notation.

Creds: http://bigocheatsheet.com/

ORDERS OF GROWTH

- Since we care about the runtime when n gets infinitely large, we can drop lower order terms and constants.
  - **$\Theta(2n^3 + 6n + \log(n)) = \Theta(n^3)$**
- Should always provide the tightest bound.

ORDERS OF GROWTH

- Count the number of iterations and/or recursive calls.
- Find the number of operations per iteration or recursive call.
- Nested loops are usually some polynomial time.
- Exponential time are usually tree recursive.
- Beware of **return** statements because it exits out of a frame before the loops are finished.

NONLOCAL

- We could only access variables in parent frames and not modify them.
- Nonlocal allows us to modify variables in parents frame and outside of the current frame.

NONLOCAL

- We could only access variables in parent frames and not modify them.
- Nonlocal allows us to modify variables in parents frame and outside of the current frame.

```
def stepper(num):
    def step():

        num = num + 1
        return num
    return step
```

NONLOCAL

- We could only access variables in parent frames and not modify them.
- Nonlocal allows us to modify variables in parents frame and outside of the current frame.

```
def stepper(num):
    def step():

        num = num + 1
        return num
    return step
```

Error: We are trying to use num before we assigned it

NONLOCAL

- We could only access variables in parent frames and not modify them.
- Nonlocal allows us to modify variables in parents frame and outside of the current frame.

```
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step
```

In **step**'s frame, does not try to find **num** in local frame.

NONLOCAL

- We could only access variables in parent frames and not modify them.
- Nonlocal allows us to modify variables in parents frame and outside of the current frame.

```
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step
```

For environment diagrams, num is not a variable in any of **step**'s frames.

NONLOCAL

- Variables in the global frame cannot be modified using **nonlocal**.
- Variables in the current frame cannot be overridden using **nonlocal**.
    - Cannot have a local and nonlocal variable with the same names. in a single frame.

```
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step
```

NONLOCAL

- What is wrong with the following code?

```
a = 5
def another_add_one():
    nonlocal a
    a += 1
another_add_one()
```

NONLOCAL

- What is wrong with the following code?

```
a = 5
def another_add_one():
    nonlocal a
    a += 1
another_add_one()
```

a is a variable in the global frame.
Nonlocal cannot be used to modify variables in the global frame.

NONLOCAL

- What is wrong with the following code?

```
def adder(x):
   def add(y):
      nonlocal x, y
      x += y
      return x
   return add
adder(2)(3)
```

NONLOCAL

- What is wrong with the following code?

```
def adder(x):
    def add(y):
        nonlocal x, y
        x += y
        return x
    return add
adder(2)(3)
```

y does not exist in any parent frames.
It is a local variable

NONLOCAL

- What is wrong with the following code?

```
def adder(x):
    z = 5
    def add(y):
        z = 8
        nonlocal x, z
        x += z
        return x
    return add
adder(2)(3)
```

RECAP

- Lists and dictionaries are mutable. Tuples and strings are immutable.
- Python list objects are references with pointers. When calling functions that takes a list, we pass in the reference (or pointer) and not create a new list.
- Orders of growth tells us how long the running time of the function approach as n approach infinity.
- Constant is better than logarithmic, which is better than polynomial, which is better than exponential.
- Lower polynomial is better than higher polynomial.
- Try drawing a call stack or tree to count the # of operations.
- Nonlocal allows modifying variables not in local frame.

APPENDIX

- Other Runtime notation
- Dictionaries

ORDERS OF GROWTH

- ***n:*** size of problem
- ***R(n)***: amount of resource used (time or space)
- $R(n) = \Theta(f(n))$
- $k_1 * f(n) \leq R(n) \leq k_2 * f(n)$
- Assume ***n*** is larger than some minimum ***m***

ORDERS OF GROWTH

- ***n:*** size of problem
- ***R(n)***: amount of resource used (time or space)
- $R(n) = \Omega(f(n))$
- $k_1 * f(n) \leq R(n)$
- Assume ***n*** is larger than some minimum ***m***

ORDERS OF GROWTH

- ***n:*** size of problem
- ***R(n)***: amount of resource used (time or space)
- $R(n) = O(f(n))$
- $R(n) \leq k_2 * f(n)$
- where k_1 and k_2 are some constants and $k_1 \leq k_2$
- Assume ***n*** is larger than some minimum ***m***

ORDERS OF GROWTH

- ***Ω(f(n))*** is a **lower** bound.
- ***O(f(n))*** is an **upper** bound.
- **Θ*(f(n))*** is a **tight** bound because it is both a lower bound and an upper bound.
  - Factorial is **O(n^2)** and **O(n)**. But the tightest bound is **O(n^2)**.

DICTIONARIES

- Dictionaries map keys to values.
- Python dictionaries are unordered.
- We can obtain a key's mapped value by indexing into the dictionary via the key.
- We can add key-value pairs anytime and can also replace a key's value with something else.

DICTIONARIES

- A dictionary key can be any immutable value.
- If we try to place an entry with a mutable key (i.e. list), we will get an unhashable type error.
- We can check whether a dictionary contains a key with in.
- However, to check if a dictionary contains a value, need to iterate through the dictionary

DICTIONARIES

```
>>> numerals = {"I" : 1, "II" : 2, "III" : 3}
>>> numerals["II"]
2
>>> numerals["IV"] = 4
>>> numerals
{"I" : 1, "II" : 2, "III" : 3, "IV" : 4}
>>> numerals["I"] = 100
>>> numerals
{"I" : 100, "II" : 2, "III" : 3, "IV" : 4}
>>> "I" in numerals
True
>>> 100 in numerals
False
```

DICTIONARIES

```
a = {"a":1, "b":2, "c":3, "d":4}
del a["a"]
{"b":2, "c":3, "d":4}
a.pop("d")
4
{"b":2, "c":3}

for key in dictionary

for key in dictionary.keys()

for value in dictionary.values()

for key, value in dictionary.items()
```

DICTIONARIES

- We can iterate over a dictionary's keys.

```
for key in dictionary

for key in dictionary.keys()
```

- We can iterate over a dictionary's values.

```
for value in dictionary.values()
```

- We can iterate over a dictionary's keys and values at the same time.

```
for key, value in dictionary.items()
```

DICTIONARIES

- We can delete a dictionary's key-value pair with **del**.

```
a = {"a":1, "b":2, "c":3, "d":4}
del a["a"]
{"b":2, "c":3, “d":4}
```

- We can delete a key and return its value with **pop**.

```
a.pop("d")
4
{"b":2, "c":3}
```