Lists

Its List o'clock

Stanford | ENGINEERING
Computer Science

Housekeeping

- Assignment 2 is released! Due Friday midnight, grace period extends to Saturday midnight

- You should be having/have had your first IGs this week for assignment 1 grading- reach out to your section leader if you don't know how to schedule IGs

Today

- **Solidify parameters and returns**
  - **The `print` statement exists!**
  - **Do factorial_avg**
  - **Introduce doctests**

- Introducing: Lists
  - A list variable
  - Traversing a list and the **`len()`** function
  - Modifying a list: indexing, **`append()`** and **`pop()`**
  - A smorgasbord of list functions
  - Lists as parameters

Moving away from images

- Most of the variables we have worked with have been pixels and images

- They can also be numbers!

- They can also be boolean values (True/False)

- They can also be any other type of value (stay tuned!)

Recall: Variables as numbers

```
def variable_num_example(filename):
  x = 3 # numbers without decimal are "ints"
  y = 8.0 # numbers with decimal are "floats"

  # math is math
  sum = x + y # sum is 11
  prod = x * y # prod is 24
  diff = x - y # diff is -5

  # regular division
  quotient = y / x # quotient is 2.6666

  # int division
  int_quotioent = y // x
  # int_quotient is 2, // truncates decimal
```

Introducing: The `print` function

- The **`print`** function takes in (as a parameter) something to print
- It displays what you passed in on the Terminal!
- Works best with simpler types, like:
  - Ints
  - Floats
  - Anything in quotes, like **`"hello"`** (we call these strings)
- **Printing is not the same as returning**

Introducing: The `print` function

- The **`print`** function takes in (as a parameter) something to print
- It displays what you passed in on the Terminal!
- Works best with simpler types, like:
  - Ints
  - Floats
  - Anything in quotes, like **`"hello"`** (we call these strings)
- **Printing is not the same as returning**

```
def variable_num_example(filename):
  x = 3
  print(x) # displays 3
  print("hello world") # displays hello world
```

factorial_average

- Let's write a function, **`factorial_average(num1, num2)`**, which takes in two numbers, prints their factorials, and then prints the average of their factorials.

- Decompose "calculate the factorial of a number" and "calculate the average of a number" into two helper functions

factorial_avg

(in pycharm)

“Changing” parameters

Lets try a “change” version of **`compute_avg`**

```
def change_num1_to_avg(num1, num2):
    num1 = (num1 + num2) / 2
```

“Changing” parameters

Lets try a “change” version of **`compute_avg`**

```
def change_num1_to_avg(num1, num2):
  num1 = (num1 + num2) / 2

def use_change_avg():
  x = 1
  y = 3
  print(x)
  change_num1_to_avg(x, y)
  print(x)
```

“Changing” parameters

Lets try a “change” version of **`compute_avg`**

```
def change_num1_to_avg(num1, num2):
    num1 = (num1 + num2) / 2

def use_change_avg():
    x = 1
    y = 3
    print(x)
    change_num1_to_avg(x, y)
    print(x)
```

Variables:

`x = 1`

`y = 3`

Console prints:

1

Stanford | ENGINEERING
Computer Science

“Changing” parameters

Lets try a “change” version of **`compute_avg`**

```
def change_num1_to_avg(num1, num2):
    num1 = (num1 + num2) / 2

def use_change_avg():
    x = 1
    y = 3
    print(x)
    change_num1_to_avg(x, y)
    print(x)
```

Variables:

`x = 1` `num1 = 1`

`y = 3` `num2 = 3`

Stanford | ENGINEERING
Computer Science

“Changing” parameters

Lets try a “change” version of **`compute_avg`**

```
def change_num1_to_avg(num1, num2):
    num1 = (num1 + num2) / 2

def use_change_avg():
    x = 1
    y = 3
    print(x)
    change_num1_to_avg(x, y)
    print(x)
```

Variables:

`x = 1` `num1 = 1.5`

`y = 3` `num2 = 3`

Stanford | ENGINEERING
Computer Science

“Changing” parameters

Lets try a “change” version of **`compute_avg`**

```
def change_num1_to_avg(num1, num2):
    num1 = (num1 + num2) / 2

def use_change_avg():
    x = 1
    y = 3
    print(x)
    change_num1_to_avg(x, y)
    print(x)
```

Variables:

`x = 1`

`y = 3`

~~`num1 = 1.5`~~

~~`num2 = 3`~~

Console prints:

???

Stanford | ENGINEERING
Computer Science

“Changing” parameters

🚫🚫The change does not persist 🚫🚫

```
def change_num1_to_avg(num1, num2):
    num1 = (num1 + num2) / 2

def use_change_avg():
    x = 1
    y = 3
    print(x)
    change_num1_to_avg(x, y)
    print(x)
```

Variables:

`x = 1`

`y = 3`

~~`num1 = 1.5`~~

~~`num2 = 3`~~

Console prints:

1

Stanford | ENGINEERING
Computer Science

“Changing” parameters

The change happens to num1, not x

```
def change_num1_to_avg(num1, num2):
    num1 = (num1 + num2) / 2
    print(num1)

def use_change_avg():
    x = 1
    y = 3
    print(x)
    change_num1_to_avg(x, y)
    print(x)
```

Variables:

`x = 1`

`y = 3`

`num1 = 1.5`

`num2 = 3`

Console prints:

1.5

Stanford | ENGINEERING
Computer Science

Why? Copies!

A copy of an int is literally a different item

Changing the copy (`num1`) won't change the original (`x`)

Variables:

```
x = 1          num1 = 1.5
y = 3          num2 = 3
```

Note: "Catching" return values

```
def factorial(num):
    result = 1
    for i in range(1, num + 1):
        result = result * i
    return result

def factorial_avg(num1, num2):
    # this is different
    factorial_1 = factorial(num1)
    # from this
    factorial(num2)
```

Note: "Catching" return values

```
def factorial(num):
    result = 1
    for i in range(1, num + 1):
        result = result * i
    return result

def factorial_avg(num1, num2):
    # the return value of factorial(num1) is saved
    # in the variable facotorial_1
    factorial_1 = factorial(num1)

    # the return value of factorial(num2) dies!
    factorial(num2) # we have no way of using it
```

Testing functions: doctests

```
def factorial(num):
    """
    This function returns the factorial of num
    Doctests:
    >>> factorial(3)
    6
    >>> factorial(0)
    1
    """
    result = 1
    for i in range(1, num + 1):
        result = result * i
    return result
```

Doctests

- In Pycharm, doctests allow you to test functions one at a time
- Right-click on the function and hit ‘run Doctest func’ to run the tests!

```
def func_name(param1, param2):
    """
    Function header comment
    >>> func_name(0, 1)
    1
    """
```

text following >>> is code to run

text without >>> is expected output of previous code

Stanford | ENGINEERING
Computer Science

Why test?

- We add doctests to our functions so we only have to debug one function at a time

- If we always had to debug an entire program, we would have a hard time knowing where to start

- Testing a helper function before using it is good practice

Code Demo:
`buggy_factorial_average`

Its buggy!

Today

- ~~Solidify parameters and returns~~
  - ~~Do factorial_avg~~
  - ~~The `print` statement exists!~~
  - ~~Introduce doctests~~

- Introducing: Lists
  - A list variable
  - Traversing a list and the **`len()`** function
  - Modifying a list: indexing, **`append()`** and **`pop()`**
  - A smorgasbord of list functions
  - Lists as parameters

Aside: Types

We have seen variables of different "Types"

- Int:

```
x_int = 5
```

- Float:

```
y_float = 3.5
```

- String:

```
color = 'green'
```

- Objects
  - SimpleImage

```
image = SimpleImage(filename)
```

  - Pixel

```
pixel = image.get_pixel(0, 0)
```

Aside: Types

We have seen variables of different "Types"

- Int:

```
x_int = 5
```

- Float:

```
y_float = 3.5
```

- String:

```
color = 'green'
```

- Objects
  - SimpleImage

```
image = SimpleImage(filename)
```

  - Pixel

```
pixel = image.get_pixel(0, 0)
```

- **New: List**
  - `list_var = [1, 2, 3]`

Stanford | ENGINEERING
Computer Science

What is a List?

- A **`list`** is way to keep track of an *ordered collection* of items
  - Items in the list are called "elements"
  - Ordered: We can refer to elements by their position
  - Collection: **`lists`** can contain multiple items
- The list dynamically adjusts its size as elements are added or removed
- Lists have a lot of built-in functionality to make using them more straightforward

Show me some `lists`!

- Creating lists
  - Lists start/end with brackets. Elements separated by commas.

```
my_list = [1, 2, 3]
reals = [4.7, -6.0, 0.22, 1.6]
strs = ['lots', 'of', 'strings', 'in', 'list']
mix = [4, 'hello', -3.2, True, 6]
empty_list = []
```

Show me some `lists`!

- List with one element is **not** the same as the element
  - Could try this out:

```
def main():
    list_one = [1]
    one = 1
    print(list_one == one)
```

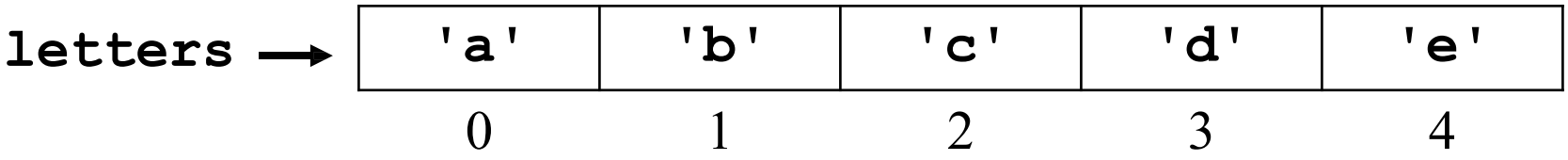
Terminal:
False

Accessing Elements of List

- Consider the following list:

```
letters = ['a', 'b', 'c', 'd', 'e']
```

- Can think of it like a series of variables that are indexed
  - Indexes start from 0

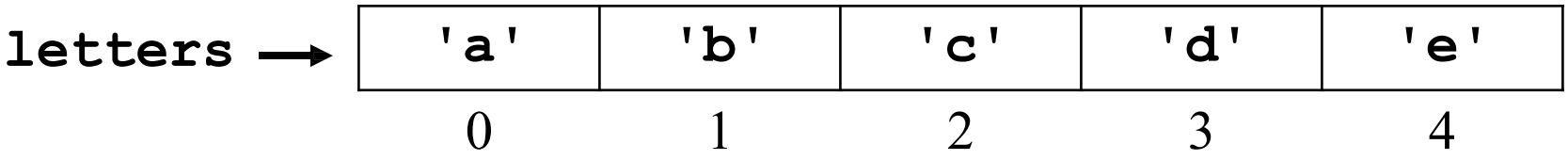


Accessing Elements of List

- Consider the following list:

```
letters = ['a', 'b', 'c', 'd', 'e']
```

- Can think of it like a series of variables that are indexed
  - Indexes start from 0



- Access individual elements:

`letters[0]` is `'a'`

`letters[4]` is `'e'`

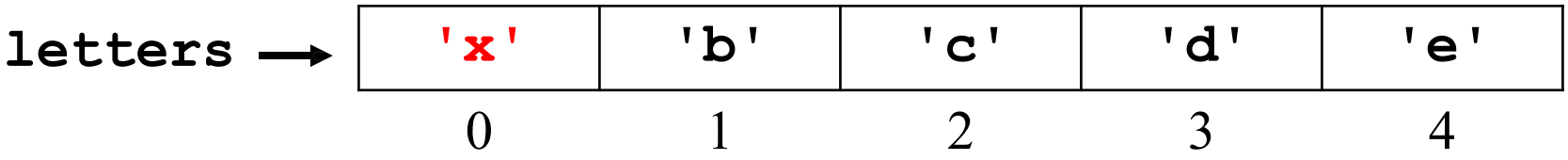
Stanford | ENGINEERING
Computer Science

Accessing Elements of List

- Consider the following list:

```
letters = ['a', 'b', 'c', 'd', 'e']
```

- Can think of it like a series of variables that are indexed
  - Indexes start from 0



- Can set individual elements like regular variable:

```
letters[0] = 'x'
```

Getting Length of a List

- Consider the following list:

```
letters = ['a', 'b', 'c', 'd', 'e']
```

- Can get length of list with **`len`** function:

```
len(letters)
```
is
```
5
```

- The last item in letters is at index **`len(letters) - 1`**

```
last_letter = letters[len(letters) - 1)
```

Iterating through a list

We can iterate through every element in a list using our handy-dandy for-loop!

```
def main():
    letters = ['a', 'b', 'c', 'd', 'e']
    for i in range(len(letters)):
        print(i, "->", letters[i])
```

Terminal:

```
0 -> a
1 -> b
2 -> c
3 -> d
4 -> e
```

Stanford | ENGINEERING
Computer Science

List Pop Quiz

- Recall our old lists:

```
my_list = [1, 2, 3]
reals = [4.7, -6.0, 0.22, 1.6]
strs = ['lots', 'of', 'strings', 'in', 'list']
mix = [4, 'hello', -3.2, True, 6]
empty_list = []
```

- Pop quiz!

```
len(my_list) = ???
len(empty_list) = ??
mix[0] = ??
strs[len(strs) - 1] = ??
empty_list[0] = ??
```

List Pop Quiz

- Recall our old lists:

```
my_list = [1, 2, 3]
reals = [4.7, -6.0, 0.22, 1.6]
strs = ['lots', 'of', 'strings', 'in', 'list']
mix = [4, 'hello', -3.2, True, 6]
empty_list = []
```

- Pop quiz!

```
len(my_list) = 3
len(empty_list) = ??
mix[0] = ??
strs[len(strs) - 1] = ??
empty_list[0] = ??
```

List Pop Quiz

- Recall our old lists:

```
my_list = [1, 2, 3]
reals = [4.7, -6.0, 0.22, 1.6]
strs = ['lots', 'of', 'strings', 'in', 'list']
mix = [4, 'hello', -3.2, True, 6]
empty_list = []
```

- Pop quiz!

```
len(my_list) = 3
len(empty_list) = 0
mix[0] = ??
strs[len(strs) - 1] = ??
empty_list[0] = ??
```

List Pop Quiz

- Recall our old lists:

```
my_list = [1, 2, 3]
reals = [4.7, -6.0, 0.22, 1.6]
strs = ['lots', 'of', 'strings', 'in', 'list']
mix = [4, 'hello', -3.2, True, 6]
empty_list = []
```

- Pop quiz!

```
len(my_list) = 3
len(empty_list) = 0
mix[0] = 4
strs[len(strs) - 1] = 'list'
empty_list[0] = ??
```

List Pop Quiz

- Recall our old lists:

```
my_list = [1, 2, 3]
reals = [4.7, -6.0, 0.22, 1.6]
strs = ['lots', 'of', 'strings', 'in', 'list']
mix = [4, 'hello', -3.2, True, 6]
empty_list = []
```

- Pop quiz!

```
len(my_list) = 3
len(empty_list) = 0
mix[0] = 4
strs[len(strs) - 1] = 'list'
empty_list[0] IndexError: list index out of range
```

Superpowered Indexing

- Can use negative index to work back from end of list

```
letters = ['a', 'b', 'c', 'd', 'e']

letters[-1] is 'e'
letters[-2] is 'd'
letters[-5] is 'a'
```

Superpowered Indexing

- Can use negative index to work back from end of list

```
letters = ['a', 'b', 'c', 'd', 'e']
```

`letters[-1]` is `'e'`

`letters[-2]` is `'d'`

`letters[-5]` is `'a'`

- For indexes, think of `-x` as the same as `len(list)-x`

  `letters[-1]` is same as `letters[len(letters)-1]`

Superpowered Indexing

- Can use negative index to work back from end of list

```
letters = ['a', 'b', 'c', 'd', 'e']
```

`letters[-1]` is `'e'`

`letters[-2]` is `'d'`

`letters[-5]` is `'a'`

- For indexes, think of `-x` as the same as `len(list)-x`

  `letters[-1]` is same as `letters[len(letters)-1]`

- How about this?

```
letters[6]
```

Superpowered Indexing

- Can use negative index to work back from end of list

```
letters = ['a', 'b', 'c', 'd', 'e']
```

`letters[-1]` is `'e'`

`letters[-2]` is `'d'`

`letters[-5]` is `'a'`

- For indexes, think of `-x` as the same as `len(list)-x`

  `letters[-1]` is same as `letters[len(letters)-1]`

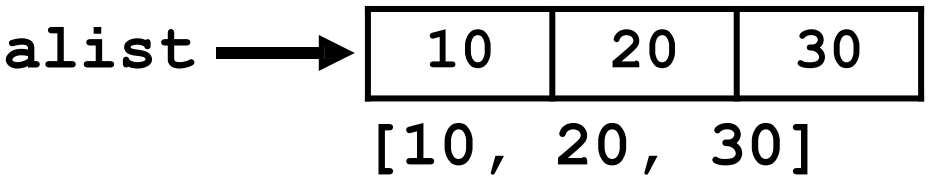
- How about this?

```
letters[6]
IndexError: list index out of range
```

Building Up Lists

- Can add elements to end of list with **`.append`**

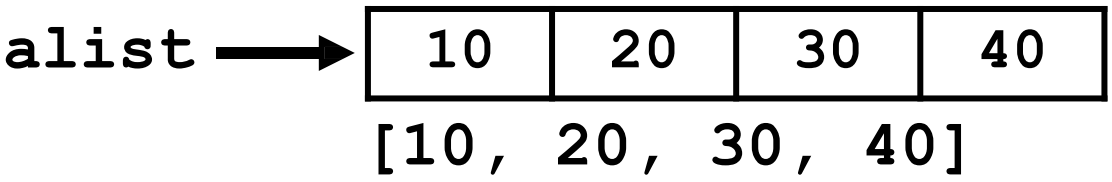
```
alist = [10, 20, 30]
```



Building Up Lists

- Can add elements to end of list with **`.append`**

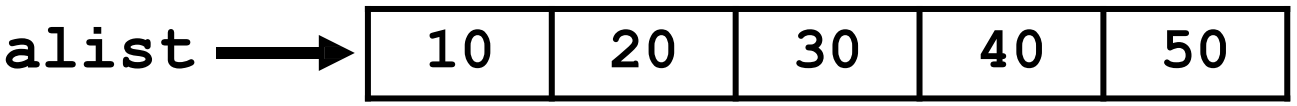
```
alist = [10, 20, 30]
alist.append(40)
```



Building Up Lists

- Can add elements to end of list with **`.append`**

```
alist = [10, 20, 30]
alist.append(40)
alist.append(50)
```

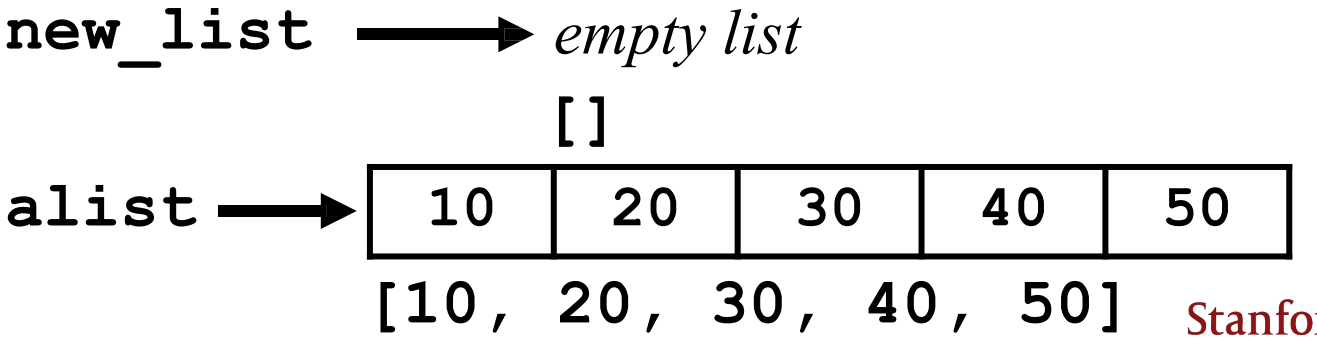


`[10, 20, 30, 40, 50]`

Building Up Lists

- Can add elements to end of list with **`.append`**

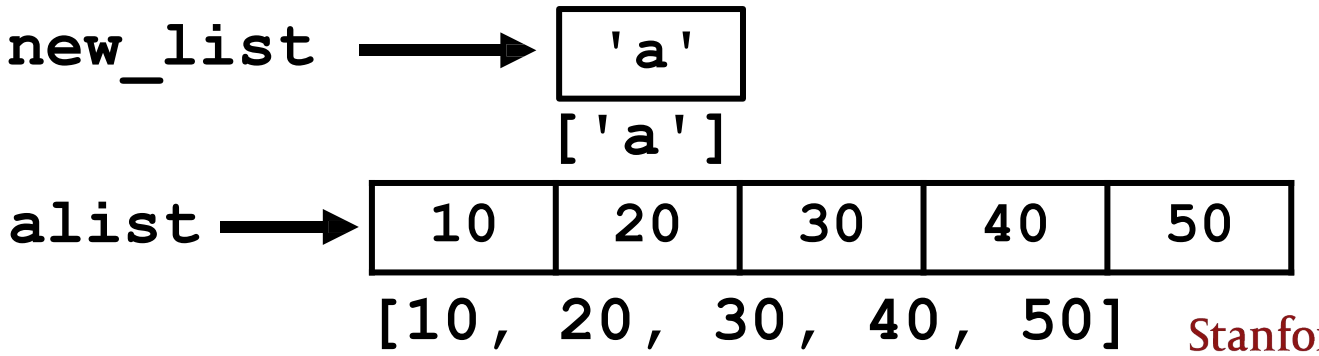
```
alist = [10, 20, 30]
alist.append(40)
alist.append(50)
new_list = []
```



Building Up Lists

- Can add elements to end of list with **`.append`**

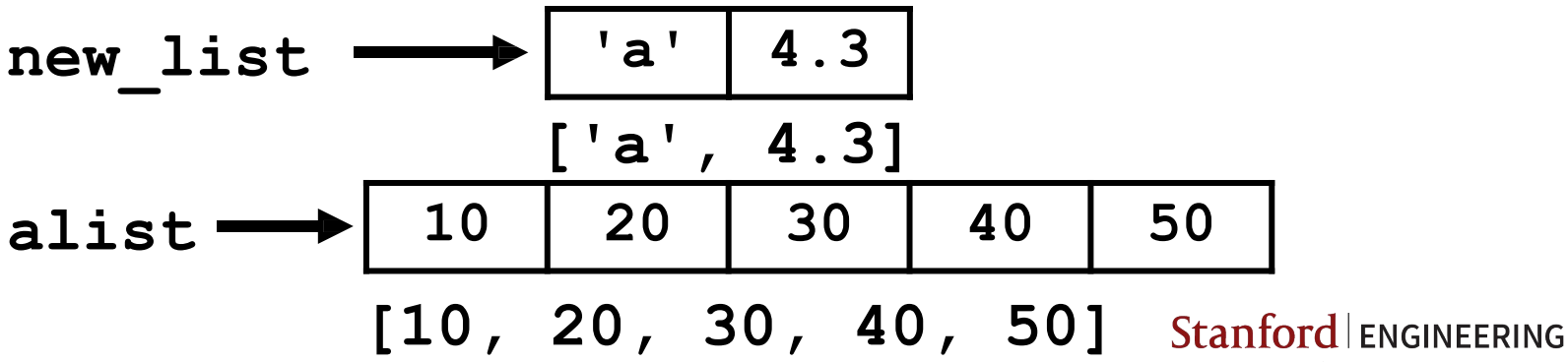
```
alist = [10, 20, 30]
alist.append(40)
alist.append(50)
new_list = []
new_list.append('a')
```



Building Up Lists

- Can add elements to end of list with **`.append`**

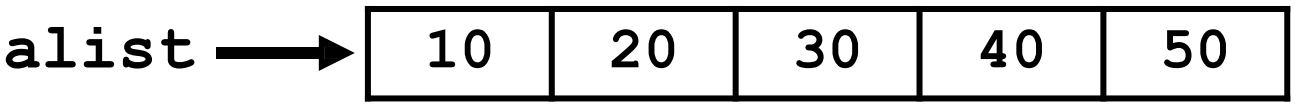
```
alist = [10, 20, 30]
alist.append(40)
alist.append(50)
new_list = []
new_list.append('a')
new_list.append(4.3)
```



Removing Elements from Lists

- Can remove elements from end of list with `.pop`
  - Removes the last element of the list and returns it

```
alist = [10, 20, 30, 40, 50]
```

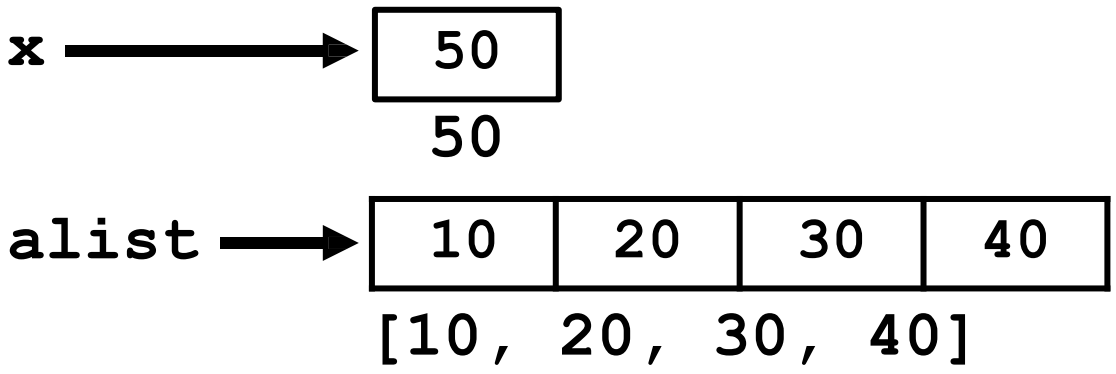


`[10, 20, 30, 40, 50]`

Removing Elements from Lists

- Can remove elements from end of list with `.pop`
  - Removes the last element of the list and returns it

```
alist = [10, 20, 30, 40, 50]
x = alist.pop()
```

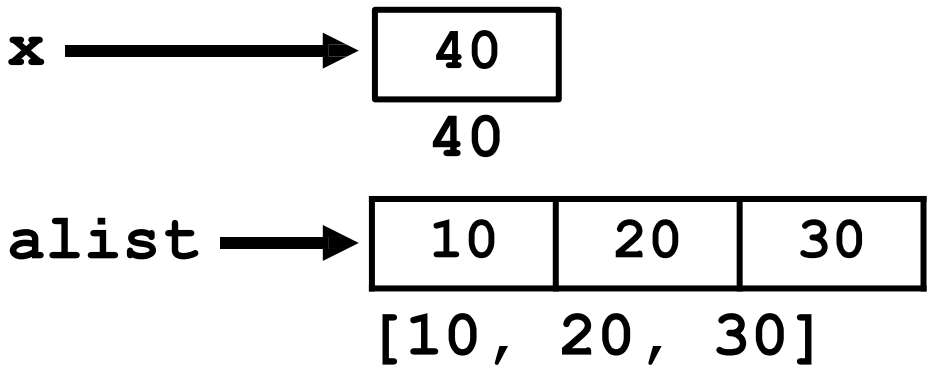


Stanford | ENGINEERING
Computer Science

Removing Elements from Lists

- Can remove elements from end of list with `.pop`
  - Removes the last element of the list and returns it

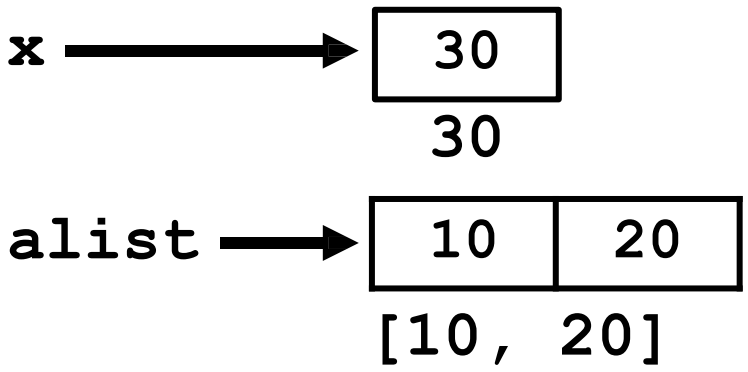
```
alist = [10, 20, 30, 40, 50]
x = alist.pop()
x = alist.pop()
```



Removing Elements from Lists

- Can remove elements from end of list with `.pop`
  - Removes the last element of the list and returns it

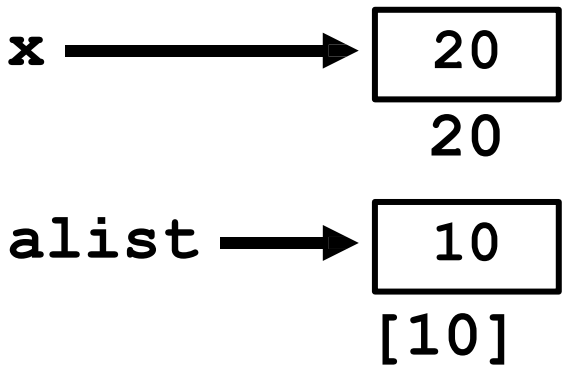
```
alist = [10, 20, 30, 40, 50]
x = alist.pop()
x = alist.pop()
x = alist.pop()
```



Removing Elements from Lists

- Can remove elements from end of list with `.pop`
  - Removes the last element of the list and returns it

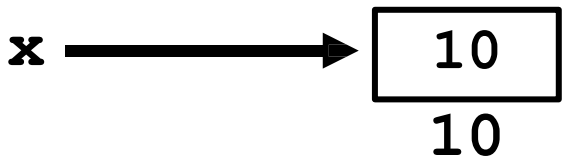
```
alist = [10, 20, 30, 40, 50]
x = alist.pop()
x = alist.pop()
x = alist.pop()
x = alist.pop()
```

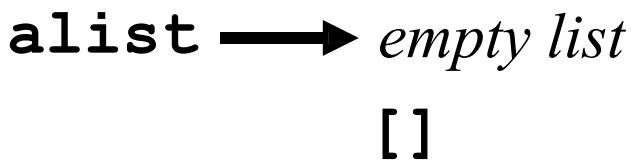


Removing Elements from Lists

- Can remove elements from end of list with `.pop`
  - Removes the last element of the list and returns it

```
alist = [10, 20, 30, 40, 50]
x = alist.pop()
x = alist.pop()
x = alist.pop()
x = alist.pop()
x = alist.pop()
```





Removing Elements from Lists

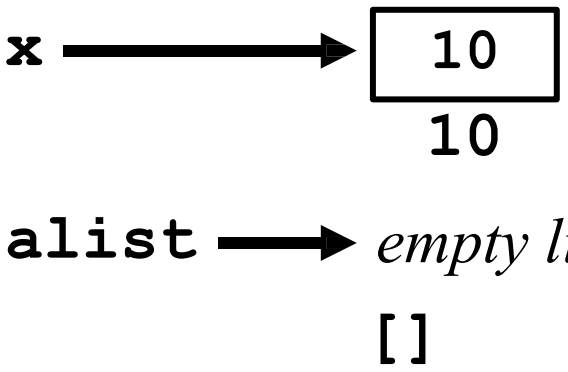
- Can remove elements from end of list with `.pop`
  - Removes the last element of the list and <u>returns it</u>

```
alist = [10, 20, 30, 40, 50]
x = alist.pop()
x = alist.pop()
x = alist.pop()
x = alist.pop()
x = alist.pop()
```

What if we did one more?

```
x = alist.pop()
```

```
IndexError: pop from empty list
```



Don't do it!

More Fun With Lists

- Can I get a couple new lists, please?

```
num_list = [1, 2, 3, 4]
str_list = ['Ruth', 'John', 'Sonia']
```

- Printing lists:

```
print(num_list)
print(str_list)
```

Terminal:

```
[1, 2, 3, 4]
['Ruth', 'John', 'Sonia']
```

More Fun With Lists

- Can I get a couple new lists, please?

```
num_list = [1, 2, 3, 4]
str_list = ['Ruth', 'John', 'Sonia']
```

- Printing lists:

```
print(num_list)
print(str_list)
```

Terminal:

```
[1, 2, 3, 4]
['Ruth', 'John', 'Sonia']
```

- Check to see if list is empty (empty list is like "False")

```
if num_list:
    print('num_list is not empty')
else:
    print('num_list is empty')
```

Even More Fun With Lists

```
num_list = [1, 2, 3, 4]
str_list = ['Ruth', 'John', 'Sonia']
```

- Check to see if a list contains an element:

```
x = 1
if x in num_list:
    # do something
```

- General form of test (evaluates to a Boolean):

```
element in list
```

  - Returns `True` if `element` is a value in `list`, `False` otherwise
  - Could use as test in a `while` loop too

Stanford | ENGINEERING
Computer Science

List Function Extravaganza (part 1)!

- Function: ***list***`.pop(index)  # pop can take parameter`
  - Removes (and returns) an element at specified index

```
fun_list = ['a', 'b', 'c', 'd']
x = fun_list.pop(2) # x will be set to 'c'
```

`fun_list` will then be `['a', 'b', 'd']`

List Function Extravaganza (part 1)!

- Function: ***list*`.pop(index)  # pop can take parameter`**
  - Removes (and returns) an element at specified index

```
fun_list = ['a', 'b', 'c', 'd']
x = fun_list.pop(2) # x will be set to 'c'
```

`fun_list` will then be `['a', 'b', 'd']`

- Function: ***list*`.remove(elem)`**
  - Removes (and returns) first occurrence of element in list

```
another_list = ['a', 'b', 'c', 'b']
another_list.remove('b')
```

  - `another_list` will then be `['a', 'c', 'b']`
  - `ValueError` if you try to remove an element that isn't in list

List Function Extravaganza (part 2)!

- Function: ***list*.extend(other_list)**
  - Adds all elements from **other_list** to list that function is called on

```
list1 = [1, 2, 3]
list2 = [4, 5]
list1.extend(list2)
```

list1 will then be **[1, 2, 3, 4, 5]**

list2 is still **[4, 5]**

List Function Extravaganza (part 2)!

- Function: ***list*`.extend(other_list)`**
  - Adds all elements from **`other_list`** to list that function is called on

```
list1 = [1, 2, 3]
list2 = [4, 5]
list1.extend(list2)
```

`list1` will then be **`[1, 2, 3, 4, 5]`**

- **`append`** is not the same as **`extend`**
  - Append adds a single element, extends merges a list onto another

```
list1 = [1, 2, 3]
list2 = [4, 5]
list1.append(list2)
```

`list1` will then be **`[1, 2, 3, [4, 5]]`**

List Function Extravaganza (part 3)!

- Using **+** operator on lists works like **`extend`** , but creates a new list. Original lists are unchanged.

```
list1 = [1, 2, 3]
list2 = [4, 5]
list3 = list1 + list2
```

`list3` will then be **`[1, 2, 3, 4, 5]`**

`list1` is still **`[1, 2, 3]`** and **`list2`** is **`[4, 5]`**

List Function Extravaganza (part 3)!

- Using **+** operator on lists works like **`extend`** , but creates a new list. Original lists are unchanged.

```
list1 = [1, 2, 3]
list2 = [4, 5]
list3 = list1 + list2
```

`list3` will then be **`[1, 2, 3, 4, 5]`**

`list1` is still **`[1, 2, 3]`** and **`list2`** is **`[4, 5]`**

- Can use **+=** operator just like **`extend`**

```
list1 += list2
```

`list1` will then be **`[1, 2, 3, 4, 5]`**

List Function Extravaganza (part 4)!

- Function: ***list*`.index(elem)`**
  - Returns index of first element in list that matches parameter elem

```
alist = ['a', 'b', 'b', 'c']
i = alist.index('b')  # i will be set to 1
```

  - **`ValueError`** if you ask for index of an element that isn't in list

List Function Extravaganza (part 4)!

- Function: ***list*.`index(elem)`**
  - Returns index of first element in list that matches parameter elem

```
alist = ['a', 'b', 'b', 'c']
i = alist.index('b')  # i will be set to 1
```

  - **`ValueError`** if you ask for index of an element that isn't in list

- Function: ***list*.`insert(index, elem)`**
  - Inserts elem at the given index. Shifts all other elements down.

```
lecturers = ['mehran', 'amrita', 'Elyse']
lecturers.insert(1, 'frankie')
```

`lecturers` will then be

```
['mehran', 'frankie', 'amrita', 'Elyse']
```

Stanford | ENGINEERING
Computer Science

Looping Through List Elements

```
str_list = ['Ruth', 'John', 'Sonia']
```

- For loop using **`range`**:

```
for i in range(len(str_list)):
    elem = str_list[i]
    print(elem)
```

Output:

```
Ruth
John
Sonia
```

Looping Through List Elements

```
str_list = ['Ruth', 'John', 'Sonia']
```

- For loop using **`range`**:

```
for i in range(len(str_list)):
    elem = str_list[i]
    print(elem)
```

- We can use a new kind of loop called a "for-each" loop

```
for elem in str_list:
    print(elem)
```

Output:

```
Ruth
John
Sonia
```

- These loops both iterate over all elements of the list
  - Variable **`elem`** is set to each value in list (in order)

Stanford | ENGINEERING
Computer Science

New loop alert

For-Each Loop Over Lists

```
str_list = ['Ruth', 'John', 'Sonia']

for elem in str_list:
    # Body of loop
    # Do something with elem
```

This code gets repeated once for each element in list

- Like variable **`i`** in **`for`** loop using **`range()`**, **`elem`** is a variable that gets updated with each loop iteration.
- **`elem`** gets assigned to each element in the list in turn.

Lists as Parameters

- When you pass a list as a parameter you are passing a reference to the actual list
  - In helper functions, changes to values in list persist after function ends (just like modifying an attribute!)

```
def add_five(num_list):
    for i in range(len(num_list)):
        num_list[i] += 5

def main():
    values = [5, 6, 7, 8]
    add_five(values)
    print(values)
```

Terminal: ???

Lists as Parameters

- When you pass a list as a parameter you are passing a reference to the actual list
  - In helper functions, changes to values in list persist after function ends (just like modifying an attribute!)

```
def add_five(num_list):
    for i in range(len(num_list)):
        num_list[i] += 5

def main():
    values = [5, 6, 7, 8]
    add_five(values)
    print(values)
```

Terminal: **[10, 11, 12, 13]**

Stanford | ENGINEERING
Computer Science

More on Lists as Parameters

- But, watch out if you create a new list in a function
  - Creating a new list means you're no longer dealing with list passed in as parameter.
  - At that point you are no longer changing parameter passed in
  - (This is moving, not modifying)

```
def create_new_list(num_list):
    num_list.append(9)
    num_list = [1, 2, 3]

def main():
    values = [5, 6, 7, 8]
    add_five(values)
    print(values)
```

Terminal: ???

More on Lists as Parameters

- But, watch out if you create a new list in a function
  - Creating a new list means you're no longer dealing with list passed in as parameter.
  - At that point you are no longer changing parameter passed in
  - (This is moving, not modifying)

```
def create_new_list(num_list):
    num_list.append(9) #modify, will persist
    num_list = [1, 2, 3] #move, wont persist

def main():
    values = [5, 6, 7, 8]
    add_five(values)
    print(values)
```

Terminal: **[5, 6, 7, 8, 9]**

Note on Loops and Lists

```
list = [10, 20, 30]
```

- For loop using **`range`**:

```
for i in range(len(list)):
    list[i] += 1 # Modifying values in list
```

- For-each loop:

```
for elem in list: # Modifying local variable
    elem += 1     # elem - NOT the value in
                  # the list, but a copy
```

Note on Loops and Lists

```
list = [10, 20, 30]
```

- For loop using **`range`**:

```
for i in range(len(list)):
    list[i] += 1 # Modifying values in list
```

- For-each loop:

```
for elem in list: # Modifying local variable
    elem += 1     # elem - NOT the value in
                  # the list, but a copy
```

- Use for loop with range when *modifying* elements of list
- Use for-each loop when ***not*** *modifying* elements of list

Put it together: `factorial_avg_list`

- Let's write a function, **`factorial_avg_list(nums)`**
  - Takes in a list of numbers and prints their factorials (as a list)
  - Then prints the average of all the factorials

- Decompose "make list of factorials" and "calculate the average of a list" into two helper functions

- (if time) Try making a "return" version of "make list of factorials" and a "modify" version

factorial_avg_list

In Pycharm!

Recap

- Lists exist! They are indexed collections of items, and there are many things we can do with them.

- A **"move" on a parameter will not persist** after the function is over, but a **"modify" will**.

- Changing elements of a list is a **"modify"**

- Most important list functions:
  - indexing (`list[index]` gets index'th item in list)
  - `len(list)`, `list.append(item)`

- The `print` function exists - it outputs text to the terminal, it is **not** the same as returning

- Doctests help us test our helper functions!