We are 183

L20: Week 12 – Wednesday

Reminders!

- Assignment 5 due Friday, April 1
- Final Project Core due a week from Friday, April 8

Last Time… on EECS 183

Interpreted Language
Comments
Dynamic and Implicit Data Types
Explicit Data Type Conversion
Handling I/O
import math
Conditionals

Python is Interpreted

```
>>> print 'Hello World!'
```

Console

```
Hello World!
```

Because python is interpreted:

- It doesn't require a **compiler**
- It can run on almost any machine
- It does this through an **interpreter**
  - which is a little slower

# Comments

```
>>> # My first Program
>>> # Author:   My Name
>>> # Date:   11-09-2015
>>> print 'Hello World!'

Hello World!

>>> 1 + 4

5

>>> # I love this – super simple
>>>
```

Multi line comments use ' ' '

```
'''
This is a multi-line comment
that continues onto a second line.
And even onto a third line.
NOTE: the first and last line are triple-quotes
You can use single or double quotes
'''
```

These are generally used to document functions, not within functions

Arithmetic Operators

Precedence	Operator	Grouping
1	( )	Left to right
2	** (exponentiation)	Right to left
3	+ - (unary), cast Example: +2, -3	Right to left
4	* / %	Left to right
5	+ - (binary) Example: 3-2	Left to right
6	=	Right to left

- Grouping defines the precedence order when several operators of the same precedence level are in an expression.

Division is similar to C++

Watch out if you have `int / int` (will floor)

```
2.0 / 3.0  ->  0.6666…

2 / 3  ->  0

5.0 / 2  ->  2.5

5 / 2  ->  2
```

Same behavior as C++

Division is similar to C++

Watch out if you have `int / int` (will floor)

```
2.0 / 3.0  ->  0.6666…

2 /

5.0

5 / 2  ->  2
```

```
-5 / 2  ->  -3
```

Python “floors” towards next negative
(C++ truncates)

Value determines data type

- The assignment determines the data type

```
age = 19  # age refers to an int

age = 5.3 # age now refers to a float
```

Data Types - boolean

- booleans only have two values

```
True
False
```

- boolean values normally are the result of comparing two values

Mixed Mode (Implicit casting)

- Mixed Data Types in expression:
  - Each sub-expression is *promoted* to the *highest* type prior to evaluation
    - In the expression **`2 * 3.5`**, the **`2`** is promoted to a `float`

- Type Promotion Guidelines
  - **`int`** is *promoted* to **`long`** is *promoted* to **`float`**

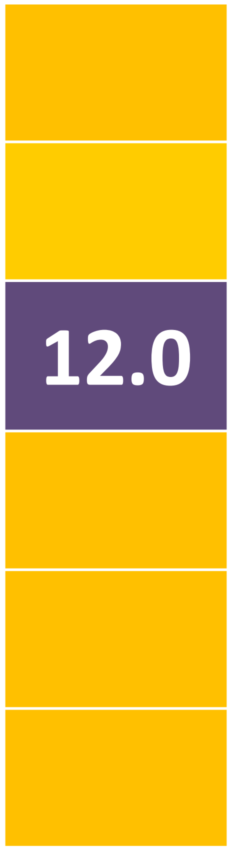
Type Conversion (Explicit casting)

What will be stored?

```
value = float(12)
```

float variable

int value



Explicit type conversion from int to float (**upcasting**)

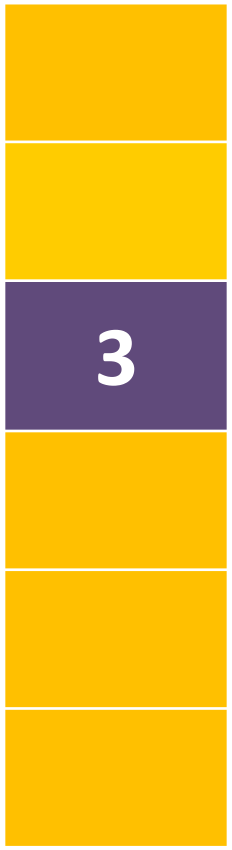
Type Conversion (Explicit casting)

What will be stored?

```
value = int(3.67)
```

int variable

float value



Explicit type conversion from float to int (**downcasting**)

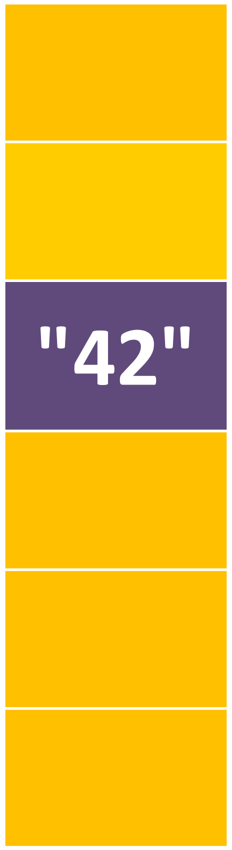
Type Conversion (Explicit casting)

What will be stored?

```
value = str(42)
```

string variable

int value



Explicit type conversion from int to string

Standard I/O Streams

- Standard Output Stream: `print`

```
print 'Hello'
```

Insertion into output stream

- Standard Input Stream: `raw_input()`

```
print 'Enter the first number:',
age = raw_input()
```

Extraction from input stream

Print multiple items – add , to suppress the line feed

```
hourlyWage = 20
print 'An hourly wage of $',
print hourlyWage, 'per hour'
print 'yields $',
print hourlyWage * 40 * 50,
print 'per year.'
```

comma

comma

Console

```
An hourly wage of $ 20 per hour
yields $ 40000 per year.
```

newline

Pythonic line continuation

```
length = int(raw_input('Enter length: '))

width = int(raw_input('Enter width: '))

print 'The area of the rectangle is:', (
    length * width )
```

Use a set of ( ) to indicate that more values are forthcoming for the above statement

Built-in Functions

int()	raw_input()	abs()
float()	print ()	min()
bool()	ord()	max()
str()	chr()	round()
type()		

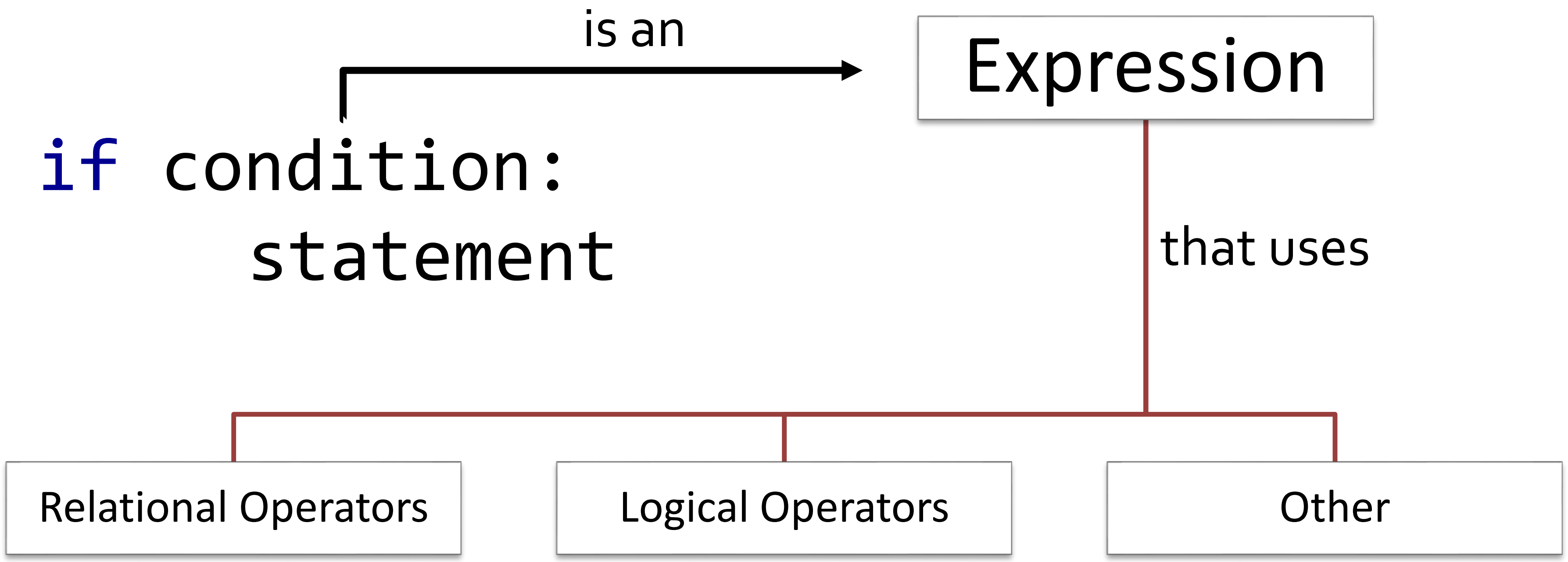
For a full list, see:
http://docs.python.org/2/library/functions.html

import math

- `math.pi`
- `math.e`
- `math.ceil(x)`
- `math.floor(x)`
- `math.fabs(x)`
- `math.pow(x,y)`
- `math.sqrt(x)`

```python
# Example
import math
print math.pi
x = math.sqrt(42)
print x
```

Conditions



```
if condition:
    statement
```

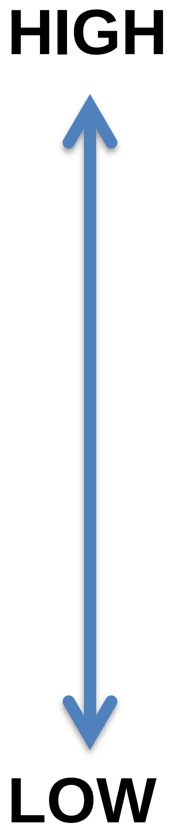
==	is equal to
!=	is not equal to
<	is smaller than
<=	is smaller than or equal to
>	is greater than
>=	is greater than or equal to

and
or
not

( )
Mathematical operators
others

Precedence Rules *Recap*

OPERATOR	***ASSOCIATIVITY***
( )	left to right
** (exponentiation)	right to left
+x -x cast	right to left
* / %	left to right
+ - (add, subtract)	left to right
< <= > >= == !=	left to right
not	left to right
and	left to right
or	left to right
=	right to left



The scope of `if`

```
someBool = True
if someBool:
    print 'This is in the if scope.'
    print 'This is ALSO in the if scope.'
    print 'Even this is in the if scope.'
print 'But this is NOT in the if scope.'
```

The scope of `if` is set by indent

Look! No Braces!!!

```
someBool = True
if someBool:
    print 'This is in the if scope.'
    print 'This is ALSO in the if scope.'
    print 'Even this is in the if scope.'
print 'But this is NOT in the if scope.'
```

The indent sets the scope of the if!

Discount books example

```
DISCOUNT = 0.30

print 'Enter list price of book: ',
price = float(raw_input())
print 'Is it used? Y or N: ',
usedCode = raw_input()

if usedCode == 'Y' or usedCode == 'y':
    print 'Applying used discount'
    price = price - (DISCOUNT * price)

print 'Selling price $', price
```

What about else?

```
DISCOUNT = 0.30

print 'Enter list price of book: ',
price = float(raw_input())
print 'Is it used? Y or N: ',
usedCode = raw_input()

if usedCode == 'Y' or usedCode == 'y':
    print 'Applying used discount'
    price = price - (DISCOUNT * price)
else:
    print 'Full price'

print 'Selling price $', price
```

Using “else if” in Python: elif

```
score = float(raw_input('Enter score: '))

if score >= 90:
    print 'Pass with an A grade'
elif score >= 80:
    print 'Pass with a B grade'
elif score >= 70:
    print 'Pass with a C grade'
else:
    print 'Not passing'
```

Multiple comparisons, same variable

- Suppose we wanted to check whether a number was in a range, like a test score
- In C++ you had to have two clauses and link them with &&
- The same thing can be done in Python:

```
if 0 <= score and score <= 100:
```

- However, Python has a shortcut that does not work in C++:

```
if 0 <= score <= 100:
```

i>Clicker #1

```
x1 = 3
x2 = 2
x3 = 1

if x1 >= x2 >= x3:
    print x1
elif x2 >= x1 >= x3:
    print x2
else:
    print x3
```

What prints?

A) 1
B) 2
C) 3
D) None of the above

i>Clicker #1

```
x1 = 3
x2 = 2
x3 = 1

if x1 >= x2 >= x3:
    print x1
elif x2 >= x1 >= x3:
    print x2
else:
    print x3
```

This test works, but the code still has a bug in it

What prints?

A) 1
B) 2
[✓ C) 3]
D) None of the above

i>Clicker #2

```
x1 = ???
x2 = ???
x3 = ???

if x1 >= x2 >= x3:
    print x1
elif x2 >= x1 >= x3:
    print x2
else:
    print x3
```

Which test case reveals the bug?

`x1, x2, x3 =`

A) `1, 2, 3`
B) `3, 3, 3`
C) `3, 1, 2`
D) `2, 3, 1`
E) None of the above

i>Clicker #2

```
x1 = ???
x2 = ???
x3 = ???

if x1 >= x2 >= x3:
    print x1
elif x2 >= x1 >= x3:
    print x2
else:
    print x3
```

All conditions must be true!

Which test case reveals the bug?

x1, x2, x3 =

A) 1, 2, 3

B) 3, 3, 3

[✓ C) 3, 1, 2]

D) 2, 3, 1

E) None of the above

Python

for loops

Loop introduction

- Python has two looping structures:
  - `while` – Loop until a condition is met
  - `for` – Loop a certain number of times

for Loop Syntax

- The for loop has this general syntax:

```
for <variable> in <container>:
```

- The <variable> is simple
  - Just make up a new variable name

- <variable> will take on the value of every element of the <container>

What is a *container*?

- The container can be <u>**any**</u> type that holds other values, such as:
  - String
  - List (up next!)
  - Tuple (after lists)
  - Dictionary
  - Other types as well

Looping through a string

```
text = 'Hello!'
for ch in text:
    print '*' + ch + '*'
```

Container

Variable

Console

```
*H*
*e*
*l*
*l*
*o*
*!*
```

i>Clicker #3

```
text = 'Test'
for ch in text:
    print ch,
```

A

```
T
e
s
t
```

B

```
Test
```

C

```
T e s t
```

D

```
T,
e,
s,
t,
```

i>Clicker #3

```
text = 'Test'
for ch in text:
    print ch,
```

Print continuation character

A
```
T
e
s
t
```

B
```
Test
```

C
```
T e s t
```

D
```
T,
e,
s,
t,
```

Python

lists

C++ array vs. Python list

- In **C++**, an **array** is the simplest **container** type
  - Declare size when created
  - **Cannot** increase in length
  - All elements of **same type**
  - Can access elements through **bracket access**
    - The name of the array, square brackets, and an index

```
arrayName[2]
```

C++ array vs. Python list

- In Python, a list is similar but several key differences
  - Starts out with **any number** of elements
  - **Can** add elements to it, increasing length
  - Elements can be **different types** from each other
    - Same list can contain int, float, string, etc. elements

- The biggest similarity:
  - Can access elements through bracket access

```
listName[2]
```

A simple list

- You can create a list by putting the values inside of square brackets, separated by commas

```
primes = [2, 3, 5, 7, 11]
```

- The print statement knows how to display a list for nice output

```
print primes
```

Console

```
[2, 3, 5, 7, 11]
```

Looping over a list

- You could output each value on a separate line using a for loop:

```
primes = [2, 3, 5, 7, 11]

for value in primes:
    print 'Prime:', value
```

Console

```
Prime: 2
Prime: 3
Prime: 5
Prime: 7
Prime: 11
```

The <variable> in <container>

```
for value in primes:
```

- Variables in Python are just names that contain references to objects

- When you run the preceding for loop,
  the variable **`value`** refers to successive members of the list **`primes`**

- You **cannot** alter **`primes`** by changing **`value`**

Trying to change value

```
primes = [2, 3, 5, 7, 11]

for value in primes:
    value = 0

print primes
```

Console

```
[2, 3, 5, 7, 11]
```

Bracket Access: Can still use `[ ]`

- Lists in Python still allow us to access elements directly through `[ ]` and an index
- But the **`for`** loop only allows us to get a value in a container but not modify it
- We need a container that has all the **indices** for the container named `primes`

  e.g., `[ 0, 1, 2, 3, 4 ]`

Creating an empty list

- There are two ways to create an empty list

- This is generally preferred (and shorter to type):

```
values = []
```

- This works also

```
values = list()
```

i>Clicker #4

- Which of the following declarations is NOT a valid list?

```
A) lst1 = []
B) lst2 = [42]
C) lst3 = [42, 99]
D) lst4 = ['Hello', 42]
E) All declarations are valid
```

i>Clicker #4

- Which of the following declarations is NOT a valid list?

```
A) lst1 = []
B) lst2 = [42]
C) lst3 = [42, 99]
D) lst4 = ['Hello', 42]
E) All declarations are valid
```

Python

ranges

How would you do this in Python?

```
for (int x = 0; x < 10; ++x) {
    cout << x << endl;
}
```

How would you do this in Python?

```
for (int x = 0; x < 10; ++x) {
    cout << x << endl;
}
```

```
for x in ???:
    print x
```

Python for loops iterate over a collection

The `range()` function

- The function named `range()` creates a list of values in the requested range

- `range(n)` creates a list of values from 0 to n-1

The `range()` function

- The function named `range()` creates a list of values in the requested range

- `range(n)` creates a list of values from 0 to n-1

```
print range(3)
```

Console

```
[0, 1, 2]
```

i>Clicker #5

- What is the LAST value printed, if the user types **93<enter>** as input?

```
n = int( raw_input('Enter number: ') )
r = range(n)
for i in r:
    print i
```

A) 0
B) 92
C) 93
D) 94
E) code won’t compile

i>Clicker #5

- What is the LAST value printed, if the user types **93<enter>** as input?

```
n = int( raw_input('Enter number: ') )
r = range(n)
for i in r:
    print i
```

A) 0
[✓ B) 92]
C) 93
D) 94
E) code won’t compile

Using range() to loop over a list

- Use `range()` and `len()` to loop over a list and allow changing of values

```
primes = [2, 3, 5, 7, 11]

for i in range( len(primes) ):
    primes[i] = 0

print primes
```

Console

```
[0, 0, 0, 0, 0]
```

Which is the "Pythonic way"?

```
# preferred, Pythonic
for value in container:
    print value

# use only if needed, such as modifying
#   the contents
for i in range( len(container) ):
    container[i] = 0
```

range() can accept two parameters

- If you give `range()` two parameters, it indicates the start value and 1 past the last value you want
- For instance:

```
>>> rng = range(10, 15)
>>> print rng
[10, 11, 12, 13, 14]
```

Python

list member functions

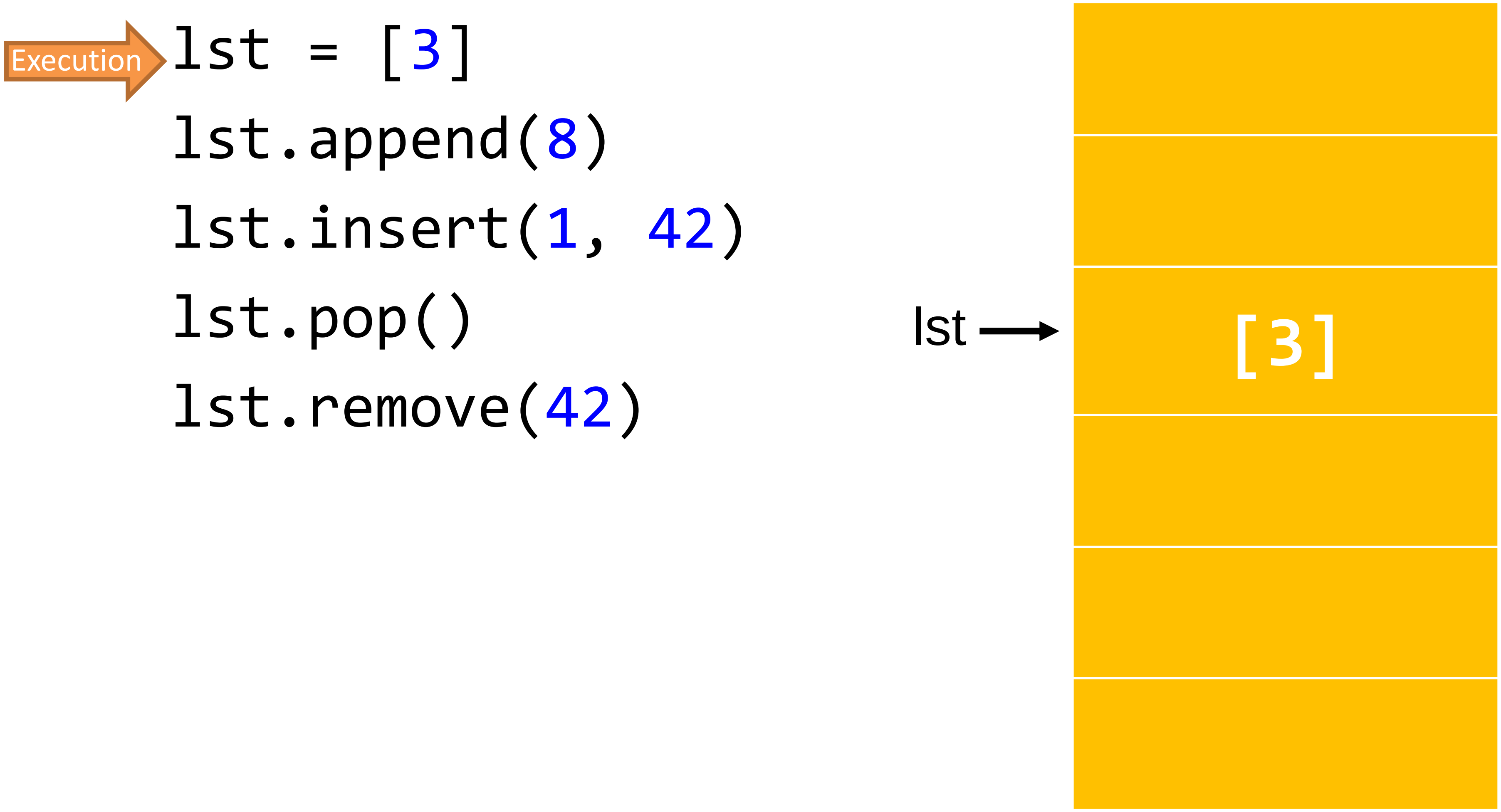
Member functions of a list

- Lists have several member functions, most of which modify the list:

```
append()    pop()

extend()

insert()    remove()

index()     count()

sort()      reverse()
```

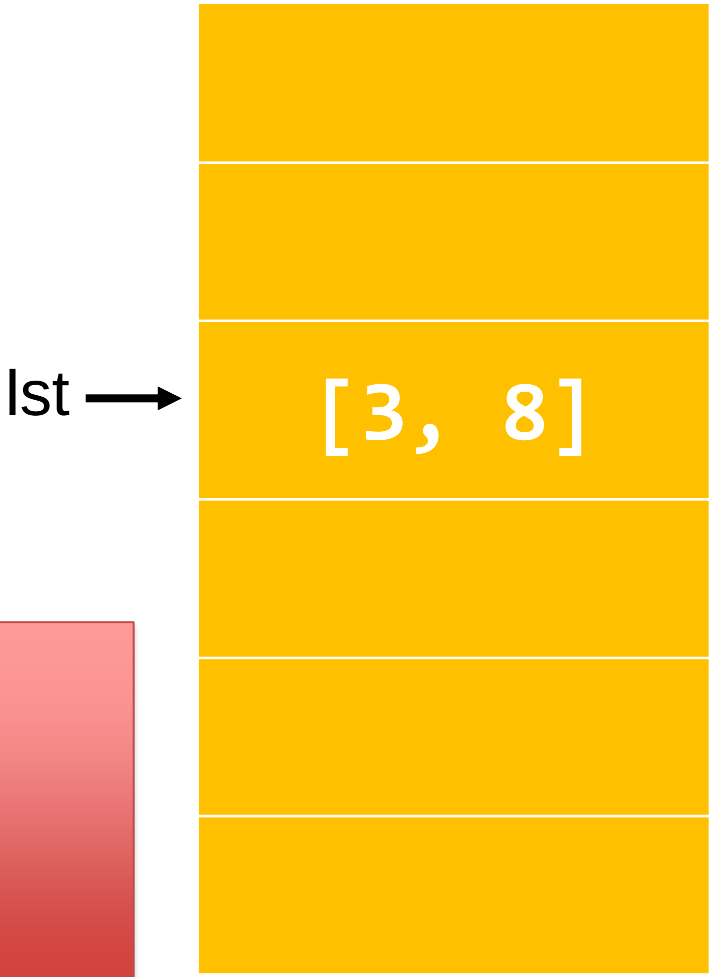
Adding and removing elements

```
lst = [3]
lst.append(8)
lst.insert(1, 42)
lst.pop()
lst.remove(42)
```



Adding at the end of the list

```
lst = [3]
lst.append(8)
lst.insert(1, 42)
lst.pop()
lst.remove(42)
```



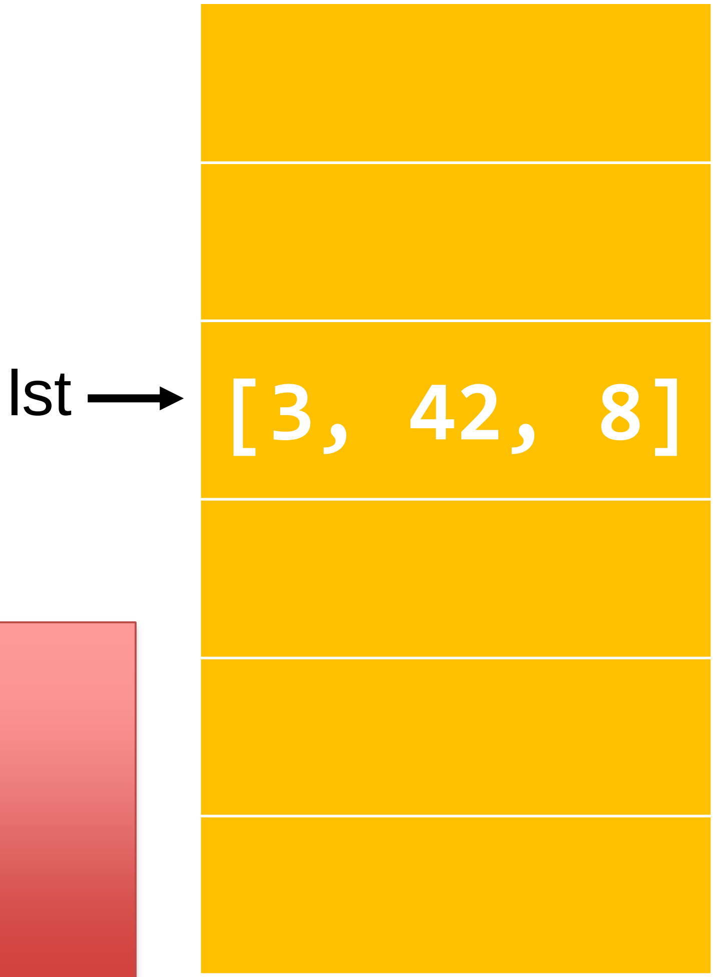
.append()

Adds an element to the end of the list

Insert 42 at index 1

Execution

```
lst = [3]
lst.append(8)
lst.insert(1, 42)
lst.pop()
lst.remove(42)
```



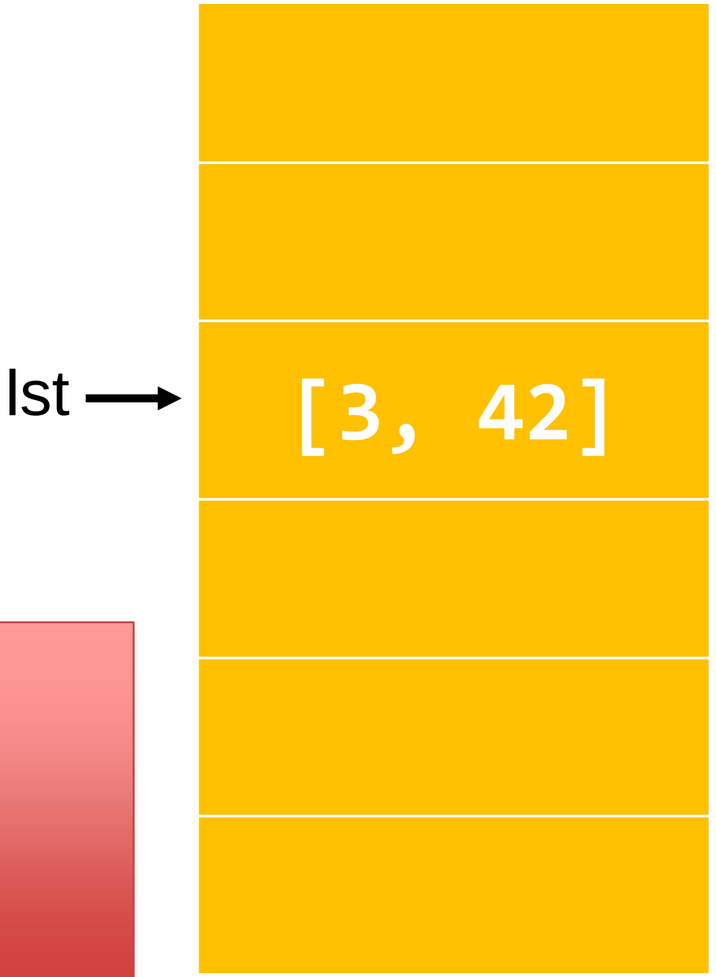
`.insert()`

Adds an element to the list at a specific point.
Shifts other elements back in the list.

Remove the “last” element

Execution

```
lst = [3]
lst.append(8)
lst.insert(1, 42)
lst.pop()
lst.remove(42)
```



.pop()

Removes an element from the end of the list.

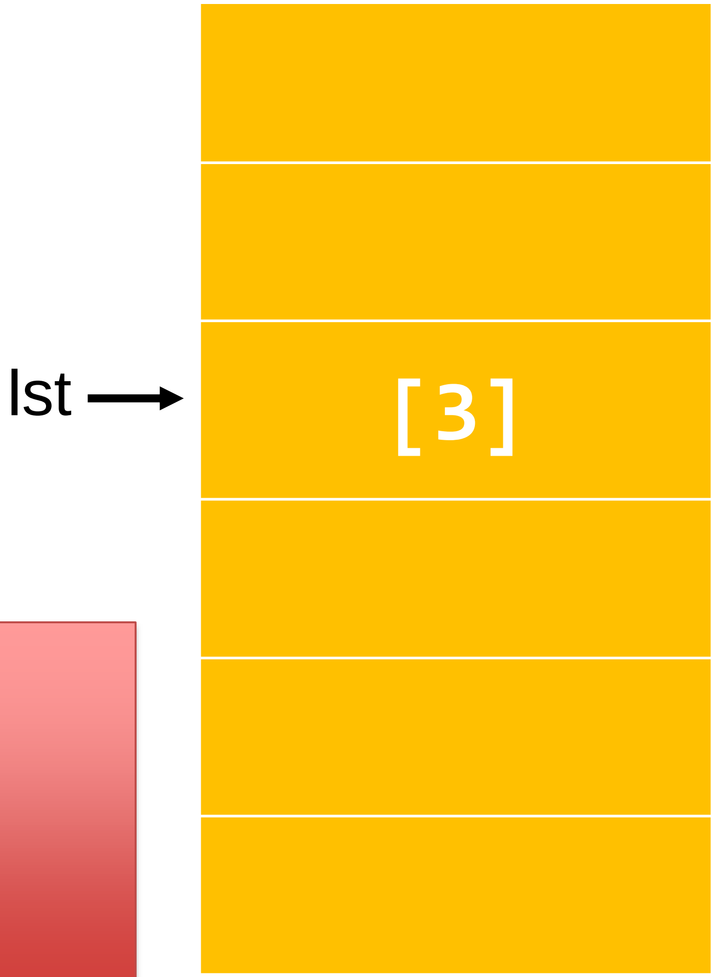
Remove the value 42 from the list

```
lst = [3]
lst.append(8)
lst.insert(1, 42)
lst.pop()
lst.remove(42)
```

Execution

.remove()

Removes a specific element from the list.
Shifts other elements forward in the list.



Notes on `.remove()`

- If duplicates exist, **only the first** is removed
  - The one with the lowest index

```
lst = [ 11, 12, 13, 12, 11 ]
lst.remove( 12 )
print lst
```

Console

```
[11, 13, 12, 11]
```

- Error thrown if value doesn’t exist in the list

```
lst.remove( 55 )
```

Console

```
ValueError: list.remove(x): x not in list
```

Finding items in a list

- Use **`.index()`** to find the **<u>first</u>** index where a value occurs (error if not in list)

```
lst = [ 11, 12, 13, 12, 11 ]
print lst.index( 11 )
```

Console

```
0
```

Counting items in a list

- The **`.count()`** member function returns how many times a value appears

```
lst = [ 11, 12, 13, 12, 11 ]
print lst.count( 12 )
```

Console

```
2
```

Counting works if item is NOT in list

- The **`.count()`** member function returns how many times a value appears

```
lst = [ 11, 12, 13, 12, 11 ]
print lst.count( 25 )
```

Console
```
0
```

Example of count() and index()

```
lst = [8, 5, 4, 8, 5]
print lst.count(8)
print lst.index(5)
print lst.count(3)
print lst.index(3)
```

Console

```
2
1
0
ValueError: 3 is not in list
```

Combining two lists

- You can concatenate two lists using **`+`** or **`.extend()`**
  - Using **`+`** produces a <u>new list</u>
  - The **`.extend()`** member function <u>modifies</u> a list

Concatenating lists

```
lst1 = [8, 5]
lst2 = [4, 8]

print lst1 + lst2
```

Console

```
[8, 5, 4, 8]
```

Concatenating lists

```
lst1 = [8, 5]
lst2 = [4, 8]

print lst1 + lst2
print lst1
```

Concatenating lst1 + lst2 **does not** change lst1 or lst2

Console

```
[8, 5, 4, 8]
[8, 5]
```

Concatenating lists

```
lst1 = [8, 5]
lst2 = [4, 8]

print lst1 + lst2
print lst1

lst3 = lst2 + lst1
print lst3
```

lst3 is a new list

Console

```
[8, 5, 4, 8]
[8, 5]
[4, 8, 8, 5]
```

Concatenating lists

```
lst1 = [8, 5]
lst2 = [4, 8]

print lst1 + lst2
print lst1

lst3 = lst2 + lst1
print lst3

lst1.extend(lst2)
print lst1
```

.extend() **does** change lst1
but **does not** change lst2

Console

```
[8, 5, 4, 8]
[8, 5]
[4, 8, 8, 5]
[8, 5, 4, 8]
```

Sorting a list

```
lst1 = [4, 8, 3, 1, 9, 0, 12, 5]
lst1.sort()
print lst1
```

Console

```
[0, 1, 3, 4, 5, 8, 9, 12]
```

Reversing a list

```
lst2 = [9, 4, 2, 7, 5, 0]
lst2.reverse()
print lst2
```

Console

```
[0, 5, 7, 2, 4, 9]
```

Sorting in reverse order

- Rather than sorting a list and then taking extra time to reverse it, you can do both at the same time

```
lst1 = [4, 8, 3, 1, 9, 0, 12, 5]
lst1.sort(reverse = True)
print lst1
```

Console

```
[12, 9, 8, 5, 4, 3, 1, 0]
```

Python

slicing

Slice: Reading only part of a sequence

- A **slice** is a way to specify a portion of a list, tuple, string, etc.

- After the name of the object, put square brackets

- Inside, put the range of indices to “extract”
  - Specify range in the form **[start : end]**

Reading only part of a sequence

- A **slice** is a way to specify a portion of a list, tuple, string, etc
- After the name of the object, put square brackets
- Inside, put the range of indices to “extract”
  - Specify range in the form **[start : end]**

First index to include

Reading only part of a list

- A **slice** is a way to specify a portion of a list, tuple, string, etc
- After the name of the object, put square brackets
- Inside, put the range of indices to “extract”
  - Specify range in the form **[start : end]**

Just past last index to include

Easier slices

- If you leave off the start or end, Python takes it to mean “that end of the list”
- So **students[1:]** means “from index 1 to the end”
- You can also use a **negative** index to mean “relative to the back end”

Examples of slices

```
text = 'Hello Python!'
print text[:5]
```

Console

```
Hello
```

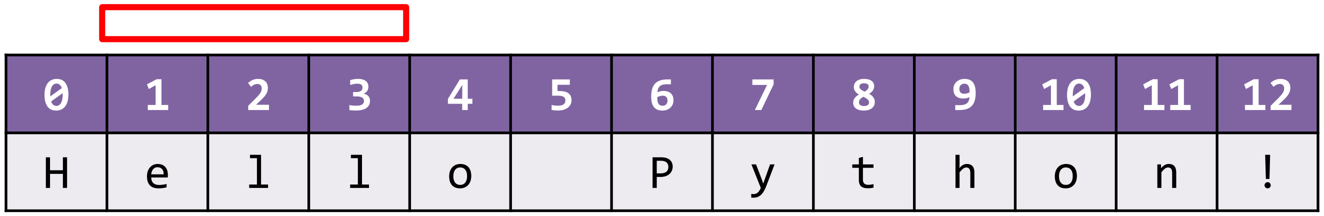
0	1	2	3	4	5	6	7	8	9	10	11	12
H	e	l	l	o		P	y	t	h	o	n	!

Examples of slices

```python
text = 'Hello Python!'
print text[:5]
print text[1:4]
```

Console

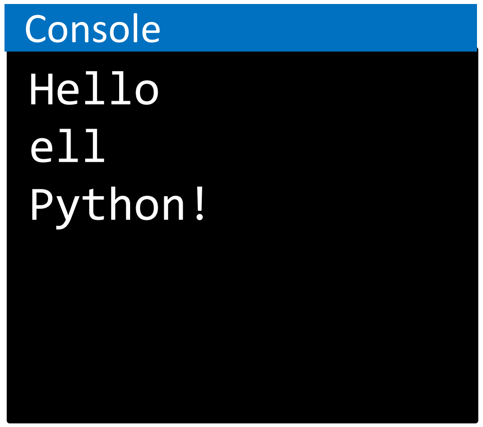
```
Hello
ell
```



0	1	2	3	4	5	6	7	8	9	10	11	12
H	e	l	l	o		P	y	t	h	o	n	!

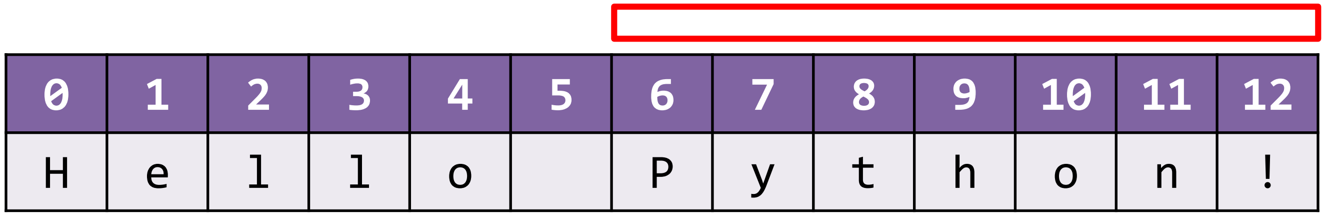
Examples of slices

```
text = 'Hello Python!'
print text[:5]
print text[1:4]
print text[6:]
```



Console

```
Hello
ell
Python!
```



0	1	2	3	4	5	6	7	8	9	10	11	12
H	e	l	l	o		P	y	t	h	o	n	!

Examples of slices

```python
text = 'Hello Python!'
print text[:5]
print text[1:4]
print text[6:]
print text[-3:-1]
```

Console

```
Hello
ell
Python!
on
```

-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
H	e	l	l	o		P	y	t	h	o	n	!

i>Clicker #6

```
text = 'Hello Python!'
print text[-1:]
```

What prints?

A) code won’t compile
B) n
C) H
D) !
E) nothing – empty string

0	1	2	3	4	5	6	7	8	9	10	11	12
H	e	l	l	o		P	y	t	h	o	n	!

i>Clicker #6

```
text = 'Hello Python!'
print text[-1:]
```

What prints?

A) code won't compile
B) n
C) H
D) !
E) nothing – empty string

0	1	2	3	4	5	6	7	8	9	10	11	12
H	e	l	l	o		P	y	t	h	o	n	!

Slice of an array

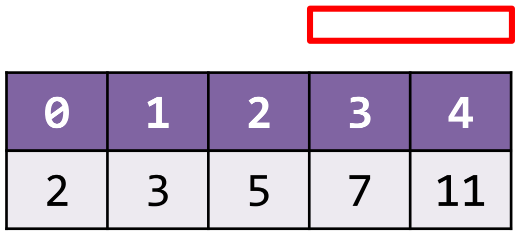
- Almost **any** operation that works on a string **also works on a list**!
  - Indexing
  - Looping over
  - Slicing

List slice example

```python
primes = [2, 3, 5, 7, 11]
print primes[3:]
```

Console

```
[7, 11]
```



0	1	2	3	4
2	3	5	7	11

List slice example

```
primes = [2, 3, 5, 7, 11]
print primes[3:]
print primes[0:3]
```

Console

```
[7, 11]
[2, 3, 5]
```

0	1	2	3	4
2	3	5	7	11

List slice example

```python
primes = [2, 3, 5, 7, 11]
print primes[3:]
print primes[0:3]
print primes[1:2]
```

Console

```
[7, 11]
[2, 3, 5]
[3]
```

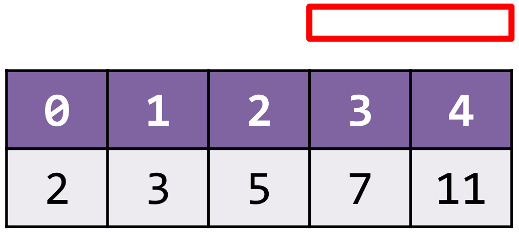
0	1	2	3	4
2	3	5	7	11

List slice example

```
primes = [2, 3, 5, 7, 11]
print primes[3:]
print primes[0:3]
print primes[1:2]
print primes[-2:]
```

Console

```
[7, 11]
[2, 3, 5]
[3]
[7, 11]
```



0	1	2	3	4
2	3	5	7	11

Python

split()

String member functions

- A string object has almost 40 different member functions!
- See

  docs.python.org/2/library/stdtypes.html#string-methods
  - or Google "python library reference string functions"

- We can use **`.split()`** to turn a string into a list

Splitting a string into a list

- The `.split()` member function breaks up a string, based on a “separator” character
  - By default, the separator is a space
    - or any amount of whitespace
  - Can specify which character to use as a parameter

- Returns a list of strings when done

Examples of .split()

```
'a b c'.split()
['a', 'b', 'c']
```

Examples of .split()

```
'a b c'.split()
['a', 'b', 'c']

'a b        c'.split()
['a', 'b', 'c']
```

Examples of .split()

```
'a b c'.split()
['a', 'b', 'c']

'a b        c'.split()
['a', 'b', 'c']

'ab cd'.split()
['ab', 'cd']
```

Examples of .split()

```
'a b c'.split()
['a', 'b', 'c']

'a b          c'.split()
['a', 'b', 'c']

'ab cd'.split()
['ab', 'cd']

'a b, cd'.split(',')
['a b', ' cd']
```

Looping over the .split() result

```
text = raw_input('Type some text: ')
print 'Splitting based on space:'

for word in text.split():
    print '*' + word + '*'
```

Console

```
Type some text: This is   fun!
Splitting based on space:
*This*
*is*
*fun!*
```

Repeated spaces only split once

Reading/summing numbers

```
sum = 0.0 # start with float
count = 0
print 'Enter a number (negative to quit):',
x = float(raw_input())

while x >= 0:
    sum += x
    count += 1 # sorry no ++ in Python
    print 'Enter a number (negative to quit):',
    x = float(raw_input())

print '\nRead', count, 'numbers, sum is:', sum
```

Reading/summing numbers With commas in line

- Now, the user will be able to enter multiple values on a single line, separated by a comma

Reading/summing numbers
Multiple numbers per line

```python
sum = 0.0
count = 0
prompt = 'Enter a number or numbers separated by commas\n'
prompt = prompt + '(just hit <Enter> to quit): '

line = raw_input(prompt)        # Input a line
while line != '':               # <Enter> will exit
    broken = line.split(',')    # Split the line
    for num in broken:
        f = float(num)          # Convert to float
        sum += f
        count += 1

    line = raw_input(prompt)    # Input another line

# Output results
print '\nRead', count, 'numbers, sum is:', sum
```