6.0001
Final Review
Type of knowledge

Declarative knowledge - a statement of fact
● Stata is building 32

Imperative knowledge - a recipe, ‘how-to’ knowledge
1. Start at Student Center
2. Walk down Mass Ave, towards Vassar St
3. Make a right on Vassar
4. Walk until you see a funky-looking building

Programming is about writing recipes to generate facts!
1) Syntax – ordering of tokens (words/characters in Python language)
   a) English Example: Noun + Verb + Adjective + Noun + Punctuation
      ■ “Colorless green ideas sleep furiously” vs. “Furiously sleep ideas colorless green”
   b) Python Example: Including necessary quotes or parentheses
      ■ print(“hello world”) vs. print “hello world”

2) Static Semantics – meaningful statements
   a. Static semantic errors happen when you put the right types of pieces in the right order, but the result has no meaning
   b. Example: 2.3/’abc’ (Syntax is correct, but does not make sense)
Expressions and Statements

**Expression** - combination of objects and operators, and can be evaluated to a value
- 3 + 5
- a or (True and b)

**Statements** - instructs the interpreter to perform some action
- Expressions
- print(3 + 5)
- return a or (True and b)
Primitive Types

1) Boolean → True, False
2) Strings → “abc”, “123”, “@#%$&@*”
3) Numbers:
   a) ints: 0, 1, 2, 3
   b) floats: 1., 1.46, 8.76, 1.1111
4) None

T/F Question:

The value of ‘math.sqrt(2.0)*math.sqrt(2.0) == 2.0’ is True.
Primitive Types

1) Boolean → True, False
2) Strings → “abc”, “123”, “@#$!$&@*”
3) Numbers:
   a) ints: 0, 1, 2, 3
   b) floats: 1., 1.46, 8.76, 1.1111
4) None

T/F Question:

The value of ‘math.sqrt(2.0)*math.sqrt(2.0) == 2.0’ is True.

2.0000000000000004 == 2.0
False
Scalar Objects

An object that **cannot** be subdivided - “the atoms of the language”

Most common Python scalar types are:

- int
- float
- Bool
- str
- NoneType
Type Issues

a. 1 // 2 = 0 (integer division)
b. 1.0 // 2 = 0.0 (integer division casted (implicitly) to float)
c. 1 / 2 = 0.5 (float division)
d. int(1 / 2) = 0 (casting)

NOTE: integer division truncates the answer – it does NOT round to nearest int (use round for that)
7 / 3 = 2.33333333
7 // 3 = 2
7 / 4 = 1.75
7 // 4 = 1
Operations

● Arithmetic operations (follow PEMDAS rules)
  ○ +, -, *, /  
  ○ ** for exponents  
  ○ % modulo to get remainder
● String operations
  ○ + for concatenation  
  ○ * to repeat
● Boolean comparators
  ○ >, >=, <, <=, ==, !=
● Logical operators
  ○ and, or, not
Swap Variables

\[
\begin{align*}
x &= 1 \\
y &= 2 \\
y &= x \\
x &= y
\end{align*}
\]

\[
\begin{align*}
x &= 1 \\
y &= 2 \\
temp &= y \\
y &= x \\
x &= temp
\end{align*}
\]
Control: IF

if condition 1:
    # some code to run
elif condition 2:
    # some other code to run instead
else:
    # some more conditions to run if the other conditions weren’t met
Control: IF

if condition 1:
    # some code to run

if condition 2:
    # other code to be run

else:
    # some code to run if condition 2 was not met
Control: Loops

for

- Repeat this block of code once per element in the given iterable

  for var in iterable:
  #code

while

- Repeat this block of code until a given condition is False

  while condition:
  #code
Control: For Loops

```python
>>> word = 'hello'
>>> for letter in word:
    print(letter)

>>> word = 'hello'
>>> for i in range(len(word)):
    print(word[i])

>>> char_list = ['a', 'b', 'c']
>>> for char in char_list:
    print(char)

>>> char_list = ['a', 'b', 'c']
>>> for i in range(len(char_list)):
    print(char_list[i])
```
Example Question

T = (0.1, 0.1)

x = 0.0

for i in range(len(T)):
    for j in T:
        x += i + j
    print (x)

print( i )

What is going to be printed?

Behind the scenes (bolded text is what is printed):
Remember, x += i + j is the equivalent of x = x + i + j

i = 0
j = 0.1 → x = x + i + j → x = 0.0 + 0 + 0.1 = 0.1
j = 0.1 → x = x + i + j → x = 0.1 + 0 + 0.1 = 0.2
i = 1
j = 0.1 → x = x + i + j → x = 0.2 + 1 + 0.1 = 1.3
j = 0.1 → x = x + i + j → x = 1.3 + 1 + 0.1 = 2.4

Last value of i was 1 → 1
Guess and Check

- Guess a value for the solution
- Check if the solution is correct
- Keep guessing until solution is good enough

Process is exhaustive enumeration, can take really long to find answer
Example of Guess & Check: Finding Square Roots

```python
number = int(input("Enter a number: ") )
answer = 0
steps = 0
while answer**2 < abs(number):
    answer = answer + 1
    steps+=1
# if square of ans is not equals to actual number, then x is not a perfect square
if answer**2 != abs(number):
    print(str(number) + ' is not a perfect square')
else:
    print('Square root of ' + str(number) + ' is ' + str(answer))
    print('The steps it took to reach the ans are: ' + str(steps))
```
Lists

- Ordered sequence of elements
- Initialized with square brackets
- Mutable

T/F Question:
Given a list \( L = ['f', 'b'] \) the statement \( L[1] = 'c' \) will mutate list \( L \).  

T/F Question:
Let \( L \) be a list, each element of which is a list of ints. In Python, the assignment statement \( L[0][0] = 3 \) mutates the list \( L \).  

```python
>>> myList = [3, 5, 2, 7]
>>> myList[0]
3
>>> myList[1] = 6
[3, 6, 2, 7]
>>> myList[:2]
[3, 6]
```
List Functions

```python
>>> letters = ['a', 'b', 'd']
>>> len(letters)
3
>>> letters.append(['e'])
['a', 'b', 'd', ['e']]
>>> letters.extend(['b', 'a'])
['a', 'b', 'd', ['e'], 'b', 'a']
>>> letters.insert(2, 'c')
['a', 'b', 'c', 'd', ['e'], 'b', 'a']
>>> letters.remove('a')
['b', 'c', 'd', ['e'], 'b', 'a']
>>> letters.reverse()
['a', 'b', ['e'], 'd', 'c', 'b']
>>> letters.pop()
'b'
>>> letters
['a', 'b', ['e'], 'd', 'c']
```
List Indexing

```python
>>> letters = ['a', 'b', 'c', 'd', 'e']
>>> letters[:]
['a', 'b', 'c', 'd', 'e']
>>> letters[2:]
['c', 'd', 'e']
>>> letters[:2]
['a', 'b']
>>> letters[:-2]
['a', 'b', 'c']
>>> letters[::2]
['a', 'c', 'e']
>>> letters[::-1]
['e', 'd', 'c', 'b', 'a']
>>> letters[1:4:2]
['b', 'd']
```
Tuples

Like lists, but immutable

t1 = (1, 2, 3, “abc”)
t2 = (5, 6, t1)

Operations:

Concatenation: t1 + t2
Indexing: (t1+t2) [3]
Slicing: (t1+t2) [1:3]

- You can iterate over tuples
- You cannot mutate tuples
- Can be used as keys in the dictionary (lists can’t) - why?
Dictionaries

- Key, value pairs
- Keys can be integers, strings, tuples, etc. (anything immutable)
- Keys can’t be lists, dictionaries, etc. (anything mutable)
- Keys are unique, values don’t have to be

**T/F Question:** In Python the keys of a dict must be immutable.  
True

**T/F Question:** The dictionary {'a':'1', 'b':'2', 'c': '3'} has a mapping of string:int  
False
Using Dictionaries

```python
>>> zoo = {'elephant': 3, 'giraffe': 4}
>>> len(zoo)
2
>>> zoo['elephant']
3
>>> zoo['frog']
KeyError: 'frog'
>>> if 'cheetah' not in zoo:
    zoo['cheetah'] = 5
>>> list(zoo.keys())
['cheetah', 'giraffe', 'elephant']
>>> list(zoo.values())
[5, 4, 3]
>>> del zoo['elephant']
>>> zoo
{'cheetah': 5, 'giraffe': 4}
```
Mutability & Aliasing

Mutable: Lists, Dictionaries
Immutable: Strings, int, float, bool, tuples, Dictionary keys

Aliasing: Two variables bound to the same object

```python
>>> a = [1]
>>> b = a
>>> a.append(2)
>>> print(a)
[1, 2]
>>> print(b)
[1, 2]
```
Mutability: Lists

L1 = ['a', 'b', 'c']
L2 = [[]], L1, 1]
L3 = [[]], ['a', 'b', 'c'], 1]
L4 = [L1]+L1
L2[1][2]='z'
print( 'L1 = ', L1 )
print( 'L2 = ', L2 )
print( 'L3 = ', L3 )
print( 'L4 = ', L4 )

What is going to be printed?

L1 = ['a', 'b', 'z']
L2 = [[]], ['a', 'b', 'z'], 1]
L3 = [[]], ['a', 'b', 'c'], 1]
L4 = [['a', 'b', 'z'], 'a', 'b', 'c']
Cloning

L1 = ['a', 'b', 'c']
L2 = L1[:]

print( 'L1 = ', L1 )
print( 'L2 = ', L2 )

L1.append('d')

print( 'L1 = ', L1 )
print( 'L2 = ', L2 )

What is going to be printed?

L1 = ['a', 'b', 'c']
L2 = ['a', 'b', 'c']
L1 = ['a', 'b', 'c', 'd']
L2 = ['a', 'b', 'c']
Abstraction & Decomposition

How to think about and solve complex systems at a high-level:

- **break up** a problem into simpler building blocks
- give each block a **name**, forget about the details of how it’s built
Abstraction & Decomposition

Why abstract and decompose?

- better code organization
- fewer lines of code
- can test small units (testing full system may be unmanageable)
Abstraction & Decomposition

How do we abstract and decompose?

Functions !!!

the most basic unit of code abstraction

Variables abstract values

Functions abstract blocks of code
Functions

1. name

def function_name(arg1, arg2, ..., argN):
    '''
    docstring here (can specify the function’s promise)
    '''
    #some code
    #some more code
    return something

2. Inputs (parameters)

3. Promises a certain behavior (if given proper inputs)
Functions

*Calling* a function ⇒ running it, with specific parameters
Functions

Calling a function ⇒ running it, with specific parameters

How to call a function:

- specify name
- *pass* the parameters
- optionally, catch the returned output

\[
\text{out} = \text{function\_name}(x_1, x_2, \ldots, x_n)
\]
Functions examples

function definition

```python
def even_or_odd(number):
    ...
    Returns True if number is even, False otherwise
    ...
    if number % 2 == 0:
        return True
    else:
        return False
```

function call

```python
three_is_even = even_or_odd(3)
```
Functions examples

def mult_by_five(number):
    print "number times 5 is ", number*5

mult_by_five("hi")

what does this do?
Functions examples

```python
def mult_by_five(number):
    print "number times 5 is ", number*5

mult_by_five("hi")
```

what does this do? Prints: hihihihihi

what is returned by `mult_by_five`? None!
Scope

scope dictates what parts of a program can see each variable's value
Scope

*scope* dictates what parts of a program can see each variable’s value

- a scope is a table, mapping variable names to values
  - assignment (<variable> = <expression>) adds an item to the table
Scope

`sacle` dictates what parts of a program can see each variable’s value

- a scope is a table, mapping variable names to values
  - assignment (`<variable> = <expression>`) adds an item to the table
- when your program starts, there’s one scope called `global scope`
Scope

`scope` dictates what parts of a program can see each variable’s value

- a scope is a table, mapping variable names to values
  - assignment (`<variable> = <expression>`) adds an item to the table
- when your program starts, there’s one scope called *global scope*
- when you call a function, a new scope is created
  - the scope is destroyed when the function returns
Scope

How is scope used?

- when a variable is used in an expression, the variable is looked up in the current scope
Scope

How is scope used?

- when a variable is used in an expression, the variable is looked up in the current scope
  - if not found, the variable is looked up in the scope where the function was defined
Scope

How is scope used?

- when a variable is used in an expression, the variable is looked up in the current scope
  - if not found, the variable is looked up in the scope where the function was defined
  - if not found there, repeat until found or we hit global scope and still not found
def testprog(x, y):
    temp = x
    x = y
    y = temp
    print(x)

x = 3
y = -3
print(x)  # 3
testprog(x, y)  # -3
print(x)  # 3

What is going to be printed?
def f(x):
    print 'In f(x): x =', x
    print 'In f(x): y =', y
    def g():
        print 'In g(): x =', x
        print( 'In g(): x =', x)
    g()

g()

x = 3
y = 2
f(1)
Recursion

a recursive function is any function that calls itself
Recursion

A recursive function is any function that calls itself.

Two crucial structural characteristics:

- **Base case**: a simplest version of the input
  - no recursive calls in base case
Recursion

a recursive function is any function that calls itself

Two crucial structural characteristics:

- **Base case**: a simplest version of the input
  - no recursive calls in base case
- **Recursive case**: makes one or more recursive calls with a simpler input
  - recursive calls **must** bring us closer to the base case
  - some basic computation is done in addition to the recursive calls
Recursion

When to use recursion?
Recursion

When to use recursion?

when a problem can be solved easily if
we have the answer to a subproblem of the same form
Recursion examples

Integer multiplication

\[ a \times b = a + a \times (b-1) \]

Factorial

\[ n! = n \times (n-1)! \]

Fibonacci

\[ \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \]
Recursion examples

Integer multiplication  
\[ a \times b = a + a \times (b-1) \]

```python
def recurMul(a, b):
    if b == 1:
        return a
    else:
        return a + recurMul(a, b-1)
```

51
Recursion examples

Factorial

\[ n! = n \times (n-1)! \]

```python
def factR(n):
    """assumes that n is an int > 0
    returns n!""
    if n == 1:
        return n
    return n*factR(n-1)
```
Recursion examples

Fibonacci

\[ \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \]

```python
def fib(x):
    """assumes x an int >= 0

    returns Fibonacci of x"
    assert type(x) == int and x >= 0
    if x == 0 or x == 1:
        return 1
    else:
        return fib(x-1) + fib(x-2)
```

Final Review
Session Part 2
Complexity

- An algorithm might be useless if it takes too long to get an answer
- We need a notion to measure how long an algorithm takes
- We would like our notion to be independent of the machine it runs on
Big O Notation

- Describes the growth of the runtime of an algorithm as a function of its input size
- Typically describes the worst case runtime
  - “In the worst case, how much time will it take for this algorithm to run?”
- When describing an algorithm, choose the tightest bound
Big-O Complexity Chart

Operations

Elements

- Horrible
- Bad
- Fair
- Good
- Excellent

- $O(n!)$
- $O(2^n)$
- $O(n^2)$
- $O(n \log n)$
- $O(n)$
- $O(\log n)$, $O(1)$
Big O Notation Mechanics

Fastest growing term dominates:

$n^2 + 100n + 1000 \log(n)$
Big O Notation Mechanics

Fastest growing term dominates:

\( n^2 + 100n + 1000 \log(n) = O(n^2) \)
Big O Notation Mechanics

Fastest growing term dominates:

\[ n^2 + 100n + 1000 \log(n) = O(n^2) \]

Constant factors do not affect complexity:

\[ 1000000000n \quad \text{vs} \quad 0.0000001n \]
Big O Notation Mechanics

Fastest growing term dominates:

\[ n^2 + 100n + 1000 \log(n) = O(n^2) \]

Constant factors do not affect complexity:

\[ 10000000000n = O(n) = 0.00000001n \]
Meaning of Complexity

- Describes how changing the size of a “large” input will affect the runtime
- If the input keeps increasing in size, eventually the algorithm with the lower complexity will be faster
- Makes no guarantee how big the input needs to get to make it faster
Complexities

$O(1)$ - Constant

$O(\log n)$ - Logarithmic

$O(n)$ - Linear

$O(n \log n)$ - Log-Linear

$O(n^k)$ - Polynomial

$O(k^n)$ - Exponential
Complexity of built-in methods

- **Constant-time, O(1)**
  - Assignment, x=2
  - Basic operations, + - * / > <

- **Dictionary**
  - Look-up: O(1)
  - Length: O(1)
  - Insert: O(1)
  - Delete: O(1)
  - `dictionary.keys()`: O(n) - because a list is generated
  - Check if a key is in the dictionary: O(1)
Complexity of built-in methods

- List
  - Append: $O(1)$
  - Length: $O(1)$
  - Insert: $O(n)$
  - Delete: $O(n)$
  - Copy: $O(n)$
  - Sort: $O(n \log n)$
  - Check if an item is in the list: $O(n)$
    - “if elt in a_list:”
Strategies for analyzing complexity

- **Loops**
  - # of iterations in the loop
  - Amount of work within each loop

- **Recursive calls**
  - # of recursive calls that are made
  - Amount of work done for each recursive call

Total Time = Time per Iteration \* # of Iterations
or Time per Call \* # of Calls
What is the complexity?

def beep(n):
    tot = 0
    while n >= 2:
        tot += n
        n = n // 2
    return tot

*Complexity: O(log n)*
def is_pal_iterate(s):
    """ input size, n = len(s) """
    string_len = len(s)
    i = 0
    while i < string_len//2 +1:
        if s[i] != s[-i-1]:
            return False
        i+=1
    return True

Number of iterations: O(n)
Number of operations in each iteration: constant

**Complexity**: O(n)
def is_pal_recursive(s):
    if len(s) == 0:
        return True
    if len(s) == 1:
        return True
    else:
        first_char = s[0]
        last_char = s[-1]
        if first_char == last_char:
            return is_pal_recursive(s[1:-1])
        else:
            return False

n/2 recursive calls: O(n)
Slicing strings: O(n)
**Complexity:** O(n^2)
Search

• Linear search
  ○ Brute force search
  ○ List doesn’t have to be sorted
  ○ $O(n)$
• Bisection search
  ○ List must be sorted to give correct answer
  ○ $O(\log n)$
Bisection search

\[ low = 0 \quad \text{high} = \text{len}(L) \]

\[ \text{guess}_1 = \frac{(low+high)}{2} \]

\[ \text{low} = 0 \quad \text{high} = \text{guess}_1 \]

\[ \text{guess}_2 = \frac{(low+high)}{2} \]

\[ \text{low} = \text{guess}_2 \quad \text{high} = \text{guess}_1 \]
Complexity of searching unsorted list

- Linear search
  - $O(n)$
  - One time search
- Bisection search
  - $\text{complexity(sort)} + \text{complexity(bisection search)}$
  - $\text{complexity(sort)} + O(\log n)$
  - $\text{complexity(sort)} > O(n)$, always
Sorting Methods: Random/Monkey Sort

- Algorithm: shuffle your list, check whether the list is sorted, if not, shuffle again
- Best case $O(n)$, already sorted, check whether list is sorted
- Worst case, unbounded
Sorting Methods: Bubble Sort

- Each step, for $i = 0, 1, \ldots, \text{len(L)}-2$, swap $L[i], L[i+1]$ such that smaller is first
- Checks every adjacent pair in list to see if it is sorted
- $n$ steps to put everything in order, building right to left
- Up to $n$ passes

<table>
<thead>
<tr>
<th>First Pass</th>
<th>Second Pass</th>
<th>Third Pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5 1 4 2 8) → (1 5 4 2 8)</td>
<td>(1 4 2 5 8) → (1 4 2 5 8)</td>
<td>(1 2 4 5 8) → (1 2 4 5 8)</td>
</tr>
<tr>
<td>(1 5 4 2 8) → (1 4 5 2 8)</td>
<td>(1 4 2 5 8) → (1 2 4 5 8)</td>
<td>(1 2 4 5 8) → (1 2 4 5 8)</td>
</tr>
<tr>
<td>(1 4 5 2 8) → (1 4 2 5 8)</td>
<td>(1 2 4 5 8) → (1 2 4 5 8)</td>
<td>(1 2 4 5 8) → (1 2 4 5 8)</td>
</tr>
<tr>
<td>(1 4 2 5 8) → (1 4 2 5 8)</td>
<td>(1 2 4 5 8) → (1 2 4 5 8)</td>
<td>(1 2 4 5 8) → (1 2 4 5 8)</td>
</tr>
</tbody>
</table>
Sorting Methods: Bubble Sort

- How many steps?
- Each step, how many operations?
- Complexity?
Sorting Methods: Bubble Sort

- How many steps? $O(n)$
- Each step, how many operations? $O(n)$
- Complexity? $O(n^2)$
Sorting Methods: Selection Sort

- Split list to prefix & suffix- prefix is sorted, suffix is unsorted
- At each step, choose the first element in suffix, add it to prefix such that prefix is still sorted
- Keep lengthening the prefix and shortening the suffix
- Build left to right

\[ 2, 3, 4, 7, 9, 6, 5, 8 \]
Sorting Methods: Selection Sort

- How many steps?
- Each step, how many operations?
- Complexity?
Sorting Methods: Selection Sort

- How many steps? $O(n)$
- Each step, how many operations? 1, 2, 3, ..., n
- Complexity? $O(n^2)$
Sorting Methods: Merge Sort

- Break list in half
- Recursively sort both halves
- Merge the sorted halves

```
mergeSort()
mergeSort()
mergeSort()
return merge(...)
return merge(...)
```

```
[4, 1, 3, 2]
[4, 1] [3, 2]
[4] [1] [3] [2]
[1, 4] [2, 3]
[1, 2, 3, 4]
```
Sorting Methods: Merge Sort

- How many levels of the recursive tree?
- How much computation of each level of the tree?
- Complexity?

```plaintext
mergeSort() [4, 1] [3, 2]
mergeSort() [4] [1] [3] [2]

return merge(...) [1, 4] [2, 3]
return merge(...) [1, 2, 3, 4]
```
Sorting Methods: Merge Sort

- How many levels of the recursive tree? $O(\log n)$
- How much computation of each level of the tree? $O(n)$
- Complexity? $O(n \log n)$
Debugging

- **Assertions**
  
  ```python
  assert <boolean condition>
  assert <boolean condition>, <argument>
  ```

- **Exception**

  ```python
  try:
  <code>
  except <exception_type>:
  <other code to run if try block encounters an exception>
  finally:
  <always executed after try, else, and except clauses>
  ```
Assertion Error

\[ x = 3 \]

\[
\text{assert } x == 4, 'x \text{ is not 4}'
\]

throws an AssertionError, stops all further computation
Exception Types

- **NameError**: access a name to a variable
  - ex. `NameError: name 'variable_name' is not defined`
- **ValueError**: concatenating a non-string with a string
- **IndexError**: accessing beyond the limits of a list
  - ex. `IndexError: list index out of range`
- **KeyError**: attempting to use a key in a dict that doesn't exist
  - ex. `KeyError: 'key_name'`
- **TypeError**: converting an inappropriate type
  - ex. `TypeError: unsupported operand type(s) for +: 'int' and 'str'`
- **AttributeError**: trying to append to a string
  - ex. `AttributeError: 'str' object has no attribute 'append'`
Input Space Partitioning

```python
def search_odd(a, b):
    """
    Inputs
    a, b: integers or booleans
    Outputs
    True if either input is equal to True or odd, and False otherwise
    """
    if type(a) == bool:
        if a:
            return True
    else:
        if a%2 == 1:
            return True
    if type(b) == bool:
        if b:
            return True
    else:
        if b%2 == 1:
            return True
    return False
```

Characterizing all types of input a program might allow based on its specification
Input Space Partitioning

```python
def search_odd(a, b):
    """
    Inputs
    a, b: integers or booleans
    Outputs
    True if either input is equal to True or odd, and False otherwise
    """
    if type(a) == bool:
        if a:
            return True
    else:
        if a%2 == 1:
            return True
    if type(b) == bool:
        if b:
            return True
    else:
        if b%2 == 1:
            return True
    return False
```

Inputs: \{bools\}, \{ints\}

Can subdivide each of those sets:

\{bools\} → \{True\}, \{False\}

\{ints\} → \{evens\}, \{odds\}
Classes

- Classes provide a means of bundling data and functionality together

- They have **attributes** and **methods** specific to itself

```python
class Vehicle(object):
    def __init__(self, name):
        self.name = name  # a variable
    def honk(self):  # a method
        print(self.name, "says HONK")
```
You can use classes to instantiate objects

```python
class Vehicle:
    def __init__(self, name):
        self.name = name  # a variable
    def honk(self):  # a method
        print(self.name, "says HONK")

>>> my_vehicle = Vehicle("batmobile")
>>> print(my_vehicle.name)
batmobile
>>> my_vehicle.honk()
batmobile says HONK
```
Classes

- **Calling class methods**

```python
class Vehicle:
    def __init__(self, name):
        self.name = name  # a variable
    def honk(self):  # a method
        print(self.name, "says HONK")

>>> my_vehicle = Vehicle("batmobile")
>>> print(my_vehicle.name)
batmobile
>>> my_vehicle.honk()
batmobile says HONK
>>> my_vehicle.honk
<bound method Vehicle.honk of <__main__.Vehicle instance at 0x1010748c0>>
```
**Classes**

- **Calling class methods**

```python
>>> my_vehicle = Vehicle("batmobile")
```

```python
>>> print(my_vehicle.name)
batmobile
```

```python
>>> my_vehicle.honk()
batmobile says HONK
```

```python
>>> my_vehicle.honk
<bound method Vehicle.honk of <__main__.Vehicle instance at 0x1010748c0>>
```
Classes

- Calling class methods

```python
>>> my_vehicle = Vehicle("batmobile")
>>> print(my_vehicle.name)
batmobile
>>> my_vehicle.honk()
batmobile says HONK
>>> my_vehicle.honk
<bound method Vehicle.honk of <__main__.Vehicle instance at 0x1010748c0>>
>>> Vehicle.honk
Vehicle.honk
<unbound method Vehicle.honk>
```
• **Calling class methods**

```python
>>> my_vehicle = Vehicle("batmobile")
>>> print(my_vehicle.name)
batmobile
>>> my_vehicle.honk()
batmobile says HONK
>>> my_vehicle.honk
<bound method Vehicle.honk of <__main__.Vehicle instance at 0x1010748c0>>
>>> Vehicle.honk
<unbound method Vehicle.honk>
```

```python
class Vehicle:
    def __init__(self, name):
        self.name = name  # a variable
    def honk(self):  # a method
        print(self.name, "says HONK")
```

what??
• Calling class methods

```python
class Vehicle:
    def __init__(self, name):
        self.name = name  # a variable
    def honk(self):  # a method
        print(self.name, "says HONK")

>>> my_vehicle = Vehicle("batmobile")
>>> print(my_vehicle.name)
batmobile
>>> my_vehicle.honk()
batmobile says HONK
>>> my_vehicle.honk
<bound method Vehicle.honk of <__main__.Vehicle instance at 0x1010748c0>>
>>> Vehicle.honk
<unbound method Vehicle.honk>
>>> Vehicle.honk()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unbound method honk() must be called with Vehicle instance as first argument (got nothing instead)
>>> Vehicle.honk(my_vehicle)
batmobile says HONK
```
Classes

- Calling class methods

```python
>>> my_vehicle = Vehicle("batmobile")
>>> print(my_vehicle.name)
batmobile
>>> my_vehicle.honk()
batmobile says HONK
>>> my_vehicle.honk
<bound method Vehicle.honk of <__main__.Vehicle instance at 0x1010748c0>>
>>> Vehicle.honk
<unbound method Vehicle.honk>
>>> Vehicle.honk()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unbound method honk() must be called with Vehicle instance as first argument (got nothing instead)
>>> Vehicle.honk(my_vehicle)
batmobile says HONK
```
You can use classes to instantiate objects

```python
>>> my_vehicle = Vehicle("batmobile")
>>> print(my_vehicle.name)
batmobile
>>> my_vehicle.honk()
batmobile says HONK
```

- **Bound method**: part of a specific object
  ```python
  >>> my_vehicle.honk
  batmobile says HONK
  >>> my_vehicle.honk()  # bound method
  batmobile says HONK
  >>> Vehicle.honk
  unbound method Vehicle.honk
  >>> Vehicle.honk()  # unbound method
  TypeError: unbound method honk() must be called with Vehicle instance as first argument (got nothing instead)
  ```
- **Unbound method**: not part of an object

```python
>>> Vehicle.honk(my_vehicle)
batmobile says HONK
```

---

**Classes**

```python
class Vehicle:
    def __init__(self, name):
        self.name = name  # a variable
    def honk(self):  # a method
        print(self.name, "says HONK")
```
Inheritance

- Let’s define a new class Car that is also a Vehicle
- ... but not all Vehicles are Cars!

```python
class Vehicle:
    def __init__(self, name):
        self.name = name  # a variable
    def honk(self):
        print(self.name, "says HONK")

class Car(Vehicle):
    def __init__(self, name):
        Vehicle.__init__(self, name)
        self.type = "car"
    def beep(self):
        print(self.name, "says BEEP")
```

- Vehicle is the **parent class (superclass)**
- Car is the **child class (subclass)**
- Car.honk exists, even if we did not explicitly define it!
- Does Vehicle.beep exist?
Inheritance

- Let’s define a new class Car that is also a Vehicle
- ... but not all Vehicles are Cars!

```python
class Car(Vehicle):
    def __init__(self, name):
        Vehicle.__init__(self, name)  # a variable
        self.type = "car"
    def beep(self):
        print(self.name, "says BEEP")
```

- Vehicle is the **parent class (superclass)**
- Car is the **child class (subclass)**
- Car.honk exists, even if we did not explicitly define it!
- Does Vehicle.beep exist? **No!**
Inheritance

- Let’s define a new class Car that is also a Vehicle
- ... but not all Vehicles are Cars!

```python
class Car(Vehicle):
    def __init__(self, name):
        Vehicle.__init__(self, name)  # a variable
        self.type = "car"
    def beep(self):
        print(self.name, "says BEEP")
```

- Vehicle is the **parent class (superclass)**
- Car is the **child class (subclass)**
- We **know** that we can treat a Car as a Vehicle if necessary, i.e. a Car is guaranteed to have all the functions a Vehicle has...
- but those functions are not guaranteed to behave the same way!
Inheritance

- Let’s define a new class Ship that is also a Vehicle
- ... and override its HONK function.

```python
class Ship(Vehicle):
    def __init__(self, name):
        Vehicle.__init__(self, name)
        self.type = "ship"
    def honk(self):
        print(self.name, "says HONK but louder")
```

- Ship.honk exists and has different behavior than Vehicle.honk
- It makes little sense to override lots of methods in the parent class
  - Not taking advantage of code reuse
  - Why not write a new independent class?
class Vehicle:
    def __init__(self, name):
        self.name = name
    def honk(self):
        print(self.name, "says HONK")

my_vehicle = Vehicle("batmobile")
my_vehicle.honk()
batmobile says HONK
Vehicle.honk(my_vehicle)
batmobile says HONK
a = my_vehicle.honk
a()
batmobile says HONK
class Car(Vehicle):
    def honk(self):
        print('hello')

my_car = Car('batmobile but better')
my_car.honk()
batmobile but better says HONK
class Car(Vehicle):
    def honk(self):
        print('hello')

my_car = Car('batmobile but better')
my_car.honk()
hello
Plotting

```python
import pylab as plt

nVals = []
linear = []
quadratic = []
cubic = []
exponential = []

for n in range(0, 30):
    nVals.append(n)
    linear.append(n)
    quadratic.append(n**2)
    cubic.append(n**3)
    exponential.append(1.5**n)

plt.plot(nVals, quadratic)
plt.show()
```
import pylab as plt

nVals = []
linear = []
quadratic = []
cubic = []
exponential = []

for n in range(0, 30):
    nVals.append(n)
    linear.append(n)
    quadratic.append(n**2)
    cubic.append(n**3)
    exponential.append(1.5**n)

plt.plot(nVals, quadratic)
plt.show()
Plotting

```
import pylab as plt

nVals = []
linear = []
quadratic = []
cubic = []
exponential = []

for n in range(0,30):
    nVals.append(n)
    linear.append(n)
    quadratic.append(n**2)
    cubic.append(n**3)
    exponential.append(1.5**n)

plt.plot(nVals, quadratic, 'r--', label='quad')
plt.plot(nVals, linear, 'k', label='linear')
plt.legend(loc='best')
plt.show()
```
Plotting

```python
import matplotlib.pyplot as plt

nVals = []
linear = []
quadratic = []
cubic = []
exponential = []

for n in range(0, 30):
    nVals.append(n)
    linear.append(n)
    quadratic.append(n**2)
    cubic.append(n**3)
    exponential.append(1.5**n)

plt.plot(nVals, quadratic, 'r--', label='quad')
plt.plot(nVals, linear, 'k', label='linear')
plt.legend(loc='best')
plt.show()
```