6.00/6.0001 Exam Review

Type of knowledge

Declarative knowledge - a statement of fact

- The square root of a number *x* is a number *y* such that $y*y = x$

Imperative knowledge - a recipe, ‘how-to’ knowledge

1. start with a guess, g
2. if g*g is close enough to x, stop and say that g is the answer
3. otherwise make a new guess by averaging g and x/g
4. Using this new guess, repeat the process until we get close enough

Rules of The Language

1) Syntax – which statements are well-formed
    a) Example: Forgetting colon after if statement or while loop, misspelling python keyword

2) Static Semantics – which statements have meaning
    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)

3) Semantics – association of each syntactically correct statement that has no semantic errors with some meaning (result may not be what programmer expected)

Expressions and Statements

Expression - combination of objects and operators, and can be evaluated to a value

- 3 + 5
- a or (True and b)

Statements - everything can make up a line and perform some action

- **Expressions**
- print('Hello')
- return

Types

1) Boolean → True, False
2) Strings → “abc”, “123”, “@#%!$&@*”
3) Numbers:
    a) ints: 0, 1, 2, 3
    b) floats: 1.46, 8.76, 1.1111

T/F Question:

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

Types

1) Boolean → True, False
2) Strings → “abc”, “123”, “@#%!$&@*”
3) Numbers:
    a) ints: 0, 1, 2, 3
    b) floats: 1.46, 8.76, 1.1111

T/F Question:

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

2.0000000000000004 == 2.0

False

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

7 / 3 = 2.33333333

7 // 3 = 2

7 / 4 = 1.75

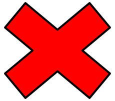
7 // 4 = 1

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

```
x = 1
y = 2
y = x
x = y
```



```
x = 1
y = 2
temp = y
y = x
x = temp
```

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: 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

```
>>> 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])

>>> |
```

Exam 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))
```

Tuples

Ordered sequence of elements

t1 = (1, 2, 3, “abc”)

t2 = (5, 6, t1)

Operations:

Concatenation: t1 + t2 `(1,2,3,'abc',5,6,(1,2,3,'abc'))`

Indexing: (t1+t2) [3] `‘abc’`

Slicing: (t1+t2) [1:3] `(1,2,3,'abc')`

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

Lists

A lot like tuples, but square brackets and can be mutated.

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

T/F Question :

Given a list L = ['f', 'b'] the statement L[1] = 'c' will mutate list L. True

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. False

List Functions

```
>>> letters = ['a','b','d']
>>> len(letters)
3
>>> letters.append('e')
['a', 'b', 'd', 'e']
>>> letters.insert(2, 'c')
['a', 'b', 'c', 'd', 'e']
>>> letters.remove('a')
['b', 'c', 'd', 'e']
```

```
>>> letters.reverse()
['e', 'd', 'c', 'b']
>>> letters.pop()
'b'
>>> letters
['e', 'd', 'c']
>>> letters.extend(['b', 'a'])
['e', 'd', 'c', 'b', 'a']
```

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

True

T/F Question: In Python the values of a dict must be immutable.

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

False

Using Dictionaries

```
>>> zoo = {'elephant' : 3, 'giraffe' : 4}
>>> len(zoo)
2
>>> zoo['elephant']
3
>>> zoo['frog']
KeyError: 'frog'
>>> if 'cheetah' not in zoo:
        zoo['cheetah'] = 5
```

```
>>> zoo
{'cheetah': 5, 'giraffe': 4, 'elephant': 3}
>>> zoo.keys()
['cheetah', 'giraffe', 'elephant']
>>> 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

```
>>> 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 )
```

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

<u>Variables</u> abstract <u>values</u>

<u>Functions</u> abstract <u>blocks of code</u>

Functions

1. name

2. Inputs (parameters)

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

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

Functions

Calling a function ⇒ executing 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

```
out = function_name(x1,x2,…,xn)
```

Functions examples

function definition

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

function call

```
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

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

```
mult_by_five("hi")
```

what does this do?

what is returned by **mult_by_five** ?

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

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

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

Exam Question

```
def testprog(x, y):
    temp = x
    x = y
    y = temp
    print(x)


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

What is going to be printed?

Scope example 1

```
def f(x):
    print 'In f(x): x =', x
    print 'In f(x): y =', y
    def g():
        print( 'In g(): x =', x)

    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 <u>crucial</u> 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

$$a*b = a + a*(b-1)$$

Factorial

$$n! = n * (n-1)!$$

Fibonacci

$$fib(n) = fib(n-1) + fib(n-2)$$

Recursion examples

Integer multiplication $a*b = a + a*(b-1)$

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

Recursion examples

Factorial

n! = n * (n-1)!

```
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 **fib(n) = fib(n-1) + fib(n-2)**

```
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)
```

Exam Review Session Part 2

https://goo.gl/B1tewp

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?”

Big O Notation Mechanics

Fastest growing term dominates:

n^2 + 100n + 1000 log(n)

Big O Notation Mechanics

Fastest growing term dominates:

$n^2 \cancel{+ 100n + 1000 \log(n)} = O(n^2)$

Big O Notation Mechanics

Fastest growing term dominates:

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

Constant factors do not affect complexity:

$1000000000n$ $0.0000001n$

Big O Notation Mechanics

Fastest growing term dominates:

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

Constant factors do not affect complexity:

$\cancel{1000000000}n = O(n) = \cancel{0.0000001}n$

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
  - Assignment
  - 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)

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):
    sum = 0
    while n >=2:
        sum +=n
        n = n // 2
    return sum
```

Complexity: O(log n)

What is the complexity?

```
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)

What is the complexity?

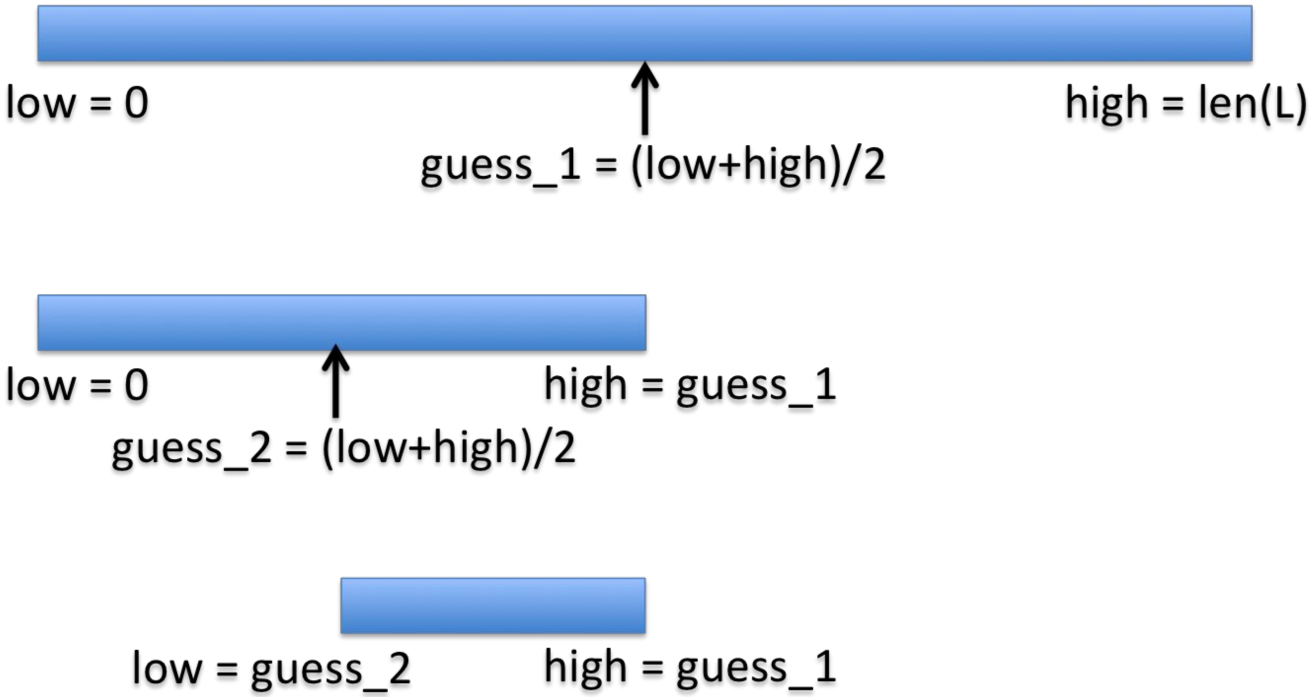
```
def is_pal_recursive(s):
    if len(s) == 0:
        return True
    if len(s) == 1:
        return True
    else:
        first_char = s[0]
        last_char = s[-1]
        if first_char == last_char:
            return is_pal_recursive(s[1:-1])
        else:
            return False
```

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



Complexity of searching unsorted list

- Linear search
  - O(n)
  - One time search
- Bisection search
  - complexity(sort) + complexity(bisection search)
  - complexity(sort) + O(log n)
  - complexity(sort) > O(n), always
- Search the same list many times, k
  - complexity(sort)/k + O(log n)
  - could be < O(n)

Complexity of sort

- 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

Complexity of sort

- Bubble sort
  - Each step, for i = 0, 1, ... , len(L)-2, swap L[i], L[i+1] such that smaller is first
  - Step 1, largest element will be at position len(L)-1
  - Step 2, second largest element will be at position len(L)-2, and so on
  - N steps to put everything in order
  - Build right to left

Step 0 -> 8 7 6 5 4
Step 1 -> 7 6 5 4 8
Step 2 -> 6 5 4 7 8
Step 3 -> 5 4 6 7 8
Step 4 -> 4 5 6 7 8
Step 5 -> 4 5 6 7 8

Complexity of sort

- Bubble sort
  - How many steps?
  - Each step, how many operations?
  - Complexity?

Step 0 -> 8 7 6 5 4
Step 1 -> 7 6 5 4 8
Step 2 -> 6 5 4 7 8
Step 3 -> 5 4 6 7 8
Step 4 -> 4 5 6 7 8
Step 5 -> 4 5 6 7 8

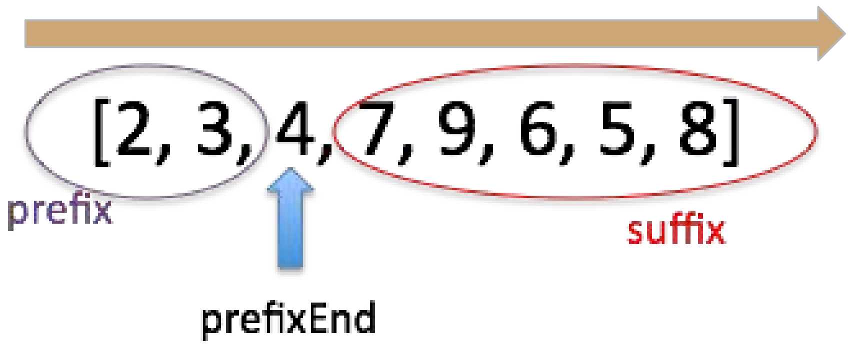
Complexity of sort

- Bubble sort
  - How many steps? O(n)
  - Each step, how many operations? O(n)
  - Complexity? $O(n^2)$

Step 0 -> 8 7 6 5 4
Step 1 -> 7 6 5 4 8
Step 2 -> 6 5 4 7 8
Step 3 -> 5 4 6 7 8
Step 4 -> 4 5 6 7 8
Step 5 -> 4 5 6 7 8

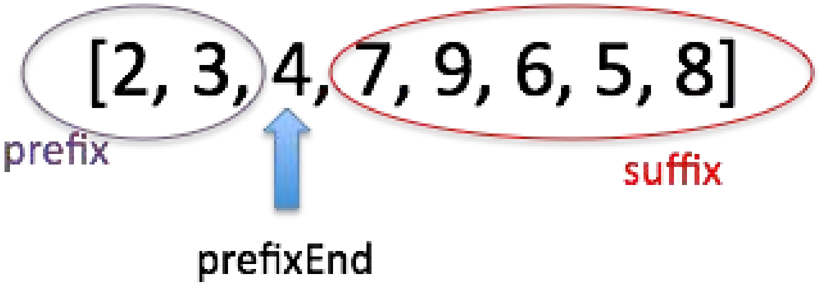
Complexity of sort

- 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



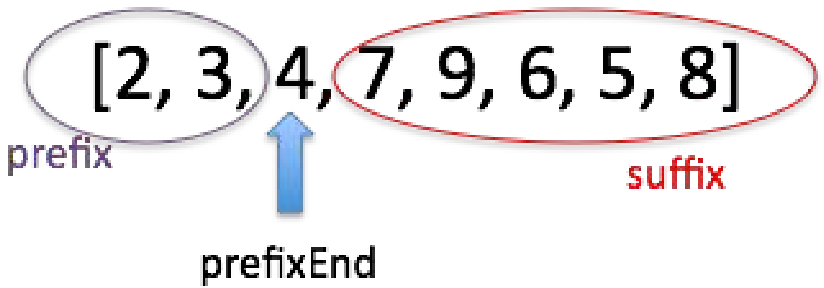
Complexity of sort

- Selection sort
  - How many steps?
  - Each step, how many operations?
  - Complexity?



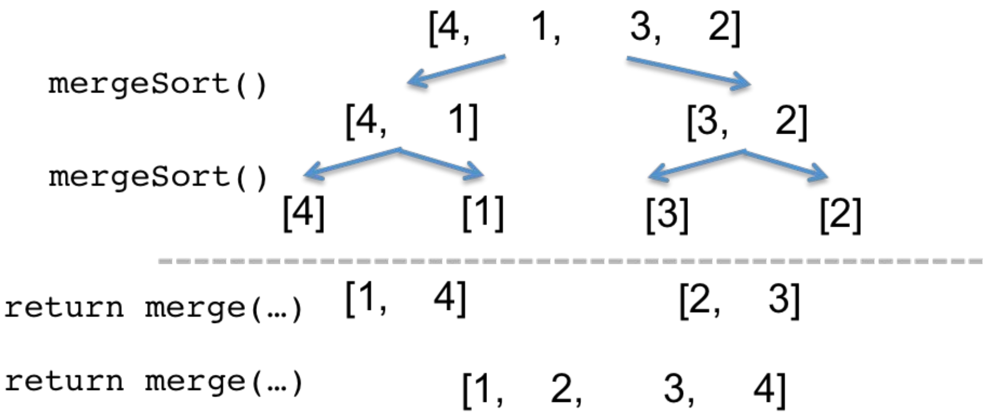
Complexity of sort

- Selection sort
  - How many steps? O(n)
  - Each step, how many operations? 1, 2, 3, ..., n
  - Complexity? $O(n^2)$



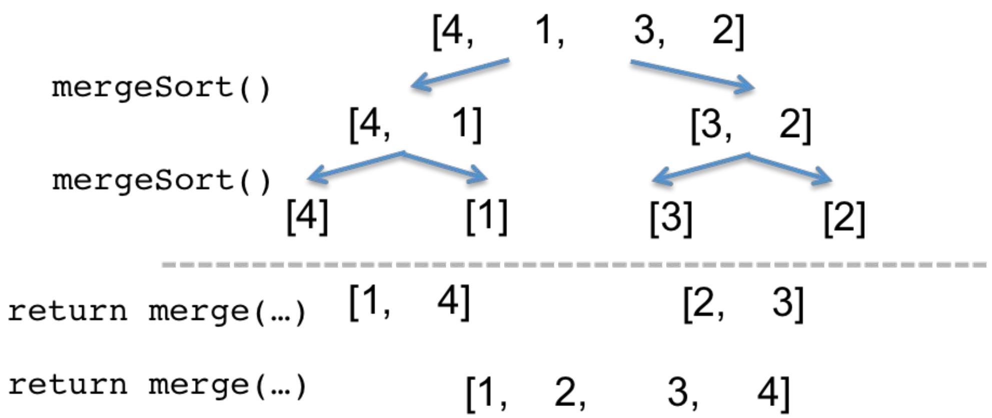
Complexity of sort

- Merge sort
  - break list in half
  - recursively sort both halves
  - merge the sorted halves



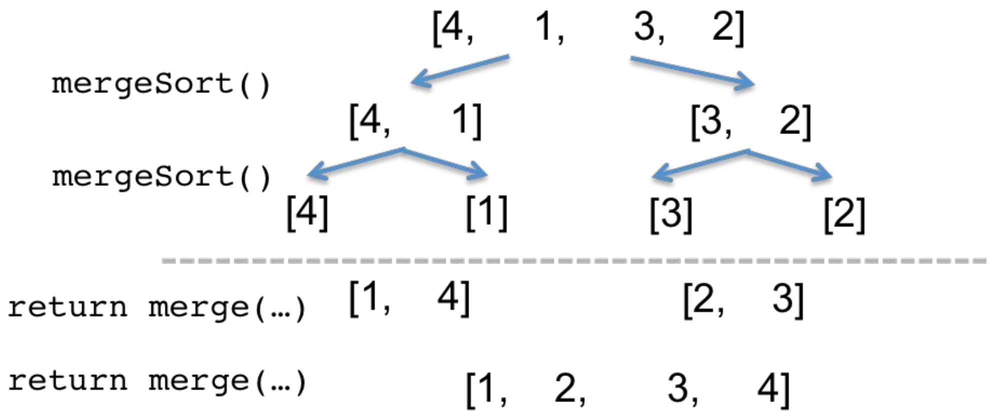
Complexity of sort

- Merge Sort
  - How many levels of the recursive tree?
  - How much computation of each level of the tree?
  - Complexity?
  - Why just count #recursive call & #computation/call?



Complexity of sort

- 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)
  - Why just count #recursive call & #computation/call? #computation/call varies across calls



Debugging

- Assertions

```
assert <boolean condition>
assert <boolean condition>, <argument>
```

- Exception

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

Classes

- Classes have **attributes** (data and procedures)

Classes

- Classes have **attributes** (data and procedures)
  (variables and methods)

Classes

- Classes have **attributes** (data and procedures)
  (variables and methods)

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

Classes

```
class Vehicle:
    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

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

Classes

```
class Vehicle:
    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

```
>>> 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

```
class Vehicle:
    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

```
>>> 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>>
```

what?

Classes

```
class Vehicle:
    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

```
>>> 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>
```

Classes

```
class Vehicle:
    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

```
>>> 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>
```

what??

Classes

```
class Vehicle:
    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

```
>>> 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

```
class Vehicle:
    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

```
>>> 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
```

WHAT

Classes

```
class Vehicle:
    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

```
>>> my_vehicle = Vehicle("batmobile")
>>> print(my_vehicle.name)
batmobile
>>> my_vehicle.honk()
batmobile says HONK
>>> my_vehicle.honk -> Bound method: part of a specific object
<bound method Vehicle.honk of <__main__.Vehicle instance at 0x1010748c0>>
>>> Vehicle.honk
<unbound method Vehicle.honk>  -> Unbound method: not part of an object
>>> 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) -> we get an error because
>>> Vehicle.honk(my_vehicle)           there is no data for the
batmobile says HONK                    method to operate on
```

Inheritance

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

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

```
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!
- Vehicle.beep • **exists • does not exist • neither**

```
>>> 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):
...     pass
...
>>> 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
>>>
```

Inheritance

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

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

```
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!
- Vehicle.beep • exists • **does not exist** • neither

Inheritance

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

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

```
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)**
- 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

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

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

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

```
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?

Hashing

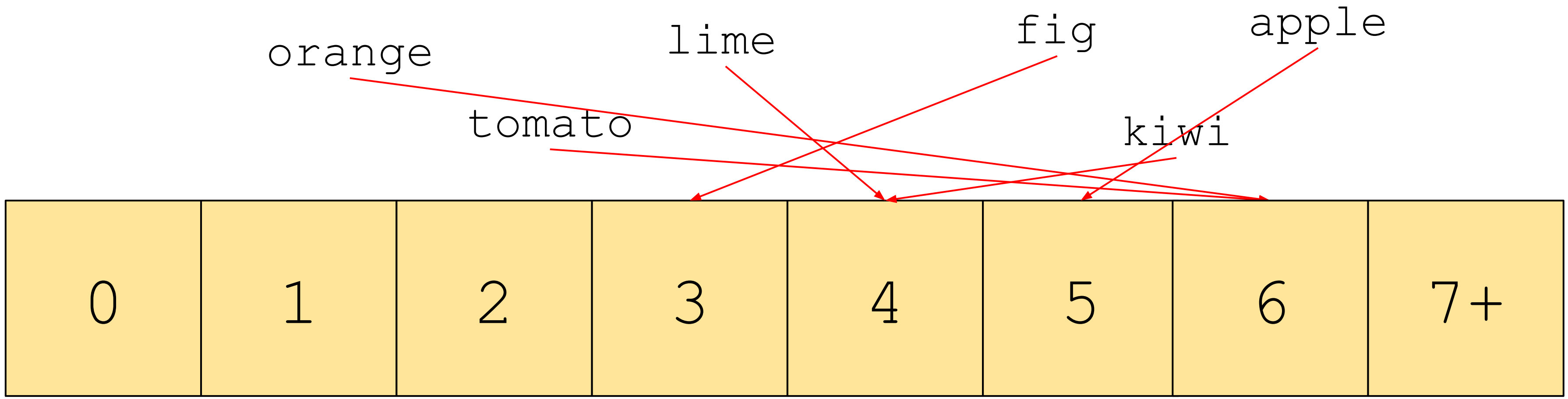
- Lists: `n` is `len(L)`
  - index O(1)
  - store O(1)
  - length O(1)
  - append O(1)
  - == O(n)
  - remove O(n)
  - copy O(n)
  - reverse O(n)
  - iteration O(n)
  - in list O(n)

- Dictionaries: `n` is `len(d)`
- worst case (very rare)
  - index O(n)
  - store O(n)
  - length O(n)
  - delete O(n)
  - iteration O(n)
- average case
  - index O(1)
  - store O(1)
  - delete O(1)
  - iteration O(n)

Why?

Hashing

- Idea: use hashtables
- Hash function: takes in a key, outputs a value in a specific range
- Simple hash function: length of string



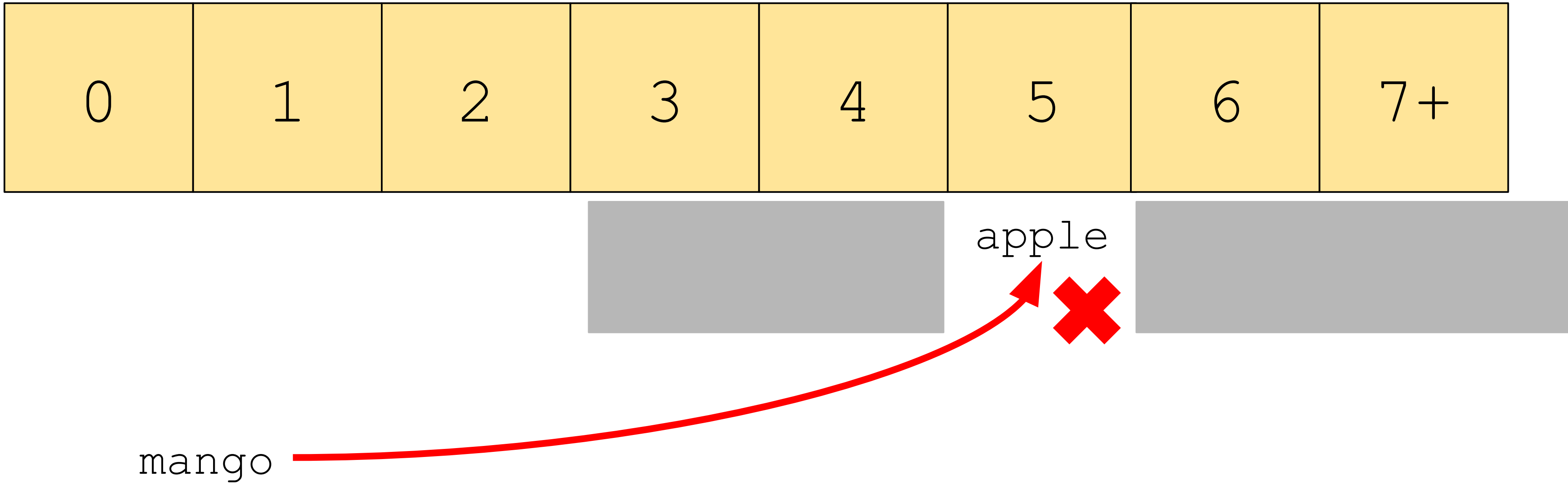
Hashing

- Lookup: much faster than lists

0	1	2	3	4	5	6	7+
			fig	kiwi	apple	orange	pomegrana
				lime		tomato	

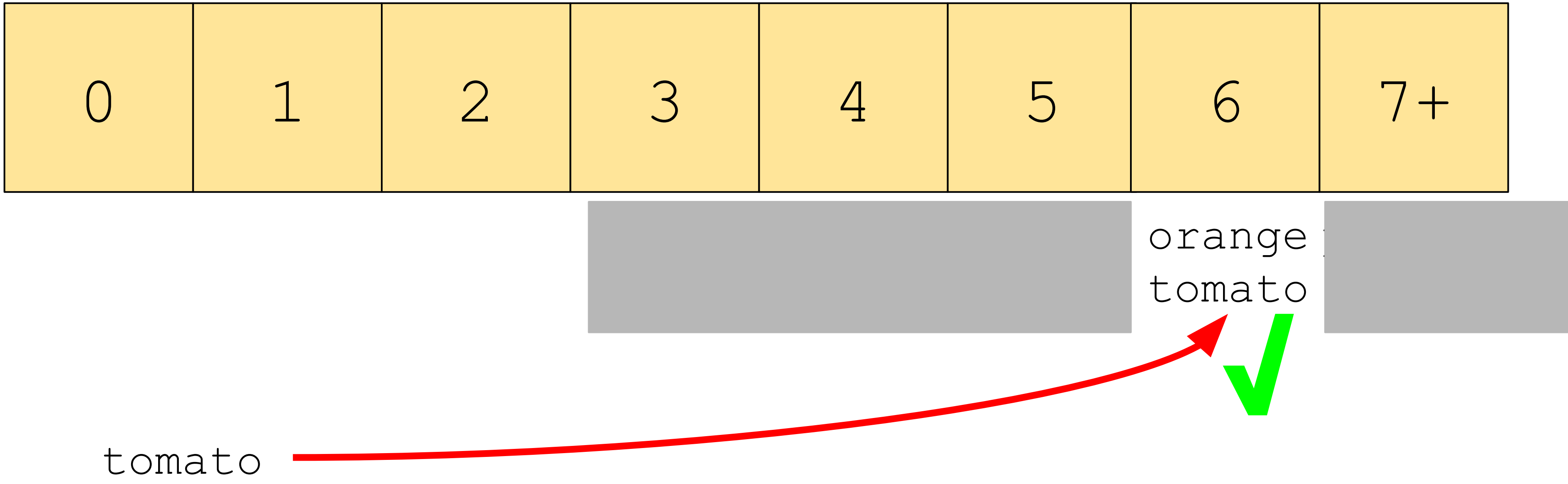
Hashing

- Lookup: much faster than lists



Hashing

- Lookup: much faster than lists



Hashing: good hashing functions

A good hashing function should

- Be deterministic, i.e. it should not use any randomness *(why?)*
- Use the entire input in the hash computation
- Should map possible inputs to outputs uniformly *(why?)*

Plotting

```python
import pylab as plt
```

Import the plotting library

```python
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)
```

Create x and y values

```python
plt.plot(nVals, quadratic)
plt.show()
```

Plot and show

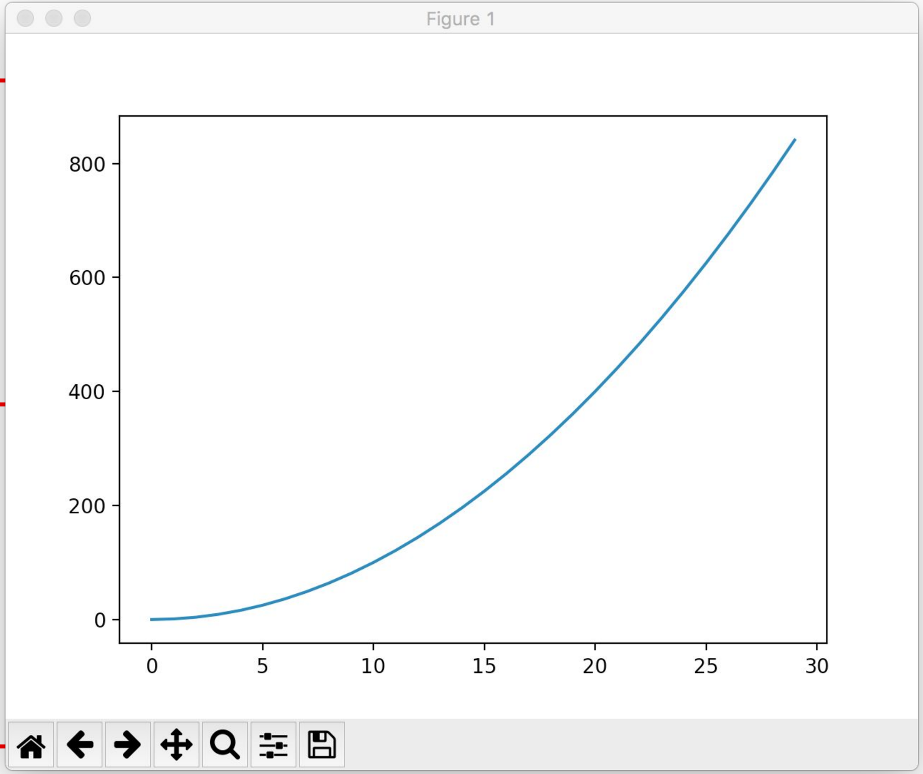
Plotting

```
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)
plt.show()
```



Plotting

```
import pylab as plt
```

Import the plotting library

```
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)
```

Create x and y values

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

Plot and 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='q
plt.plot(nVals, linear, 'k', label='linea
plt.legend(loc='best')
plt.show()
```