6.100A
Final Review

Ashhad Alam, Matt Feng, Esha Ranade

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!

Rules of The Language

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)`

Statement - instructs the interpreter to perform some action

- `print(3 + 5)`
- `return a or (True and b)`

Immutable built-in Types

1) Booleans: `True`, `False`
2) Strings: `"abc"`, `"123"`, `"@#%!$&@*"`
3) Numbers:
   a) `ints`: `-3`, `0`, `5`, `1374829`,
   b) `floats`: `1.`, `1.46`, `8.76`, `1.1111`
4) `None`

Type Issues

`1 / 2` = 0.5	float division
`int(1 / 2)` = 0	casting to `int`, i.e. truncating
`1 // 2` = 0	integer division
`1.0 // 2` = 0.0	integer division cast (implicitly) to `float`

Integer division `floor`s the answer – it does NOT truncate (`int`), nor `round`

`-7 / 3` = -2.33333333
`-7 // 3` = -3
`int(-7 / 3)` = -2

`7 / 3` = 2.33333333
`7 // 3` = 2
`int(7 / 3)` = 2

Operations

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

Representing Numbers - Binary

Integers are represented in **base 2**, also known as binary:

$$1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 = 181$$

$$2^7 \quad 2^6 \quad 2^5 \quad 2^4 \quad 2^3 \quad 2^2 \quad 2^1 \quad 2^0$$

$$128 + 0 + 32 + 16 + 0 + 4 + 0 + 1$$

Representing Numbers - Floats

Two parts, the **mantissa** and the **exponent**, both represented in **binary**

$$M * 2^{p} \rightarrow (\underbrace{M}_{\text{mantissa}}, \underbrace{p}_{\text{exponent}})$$

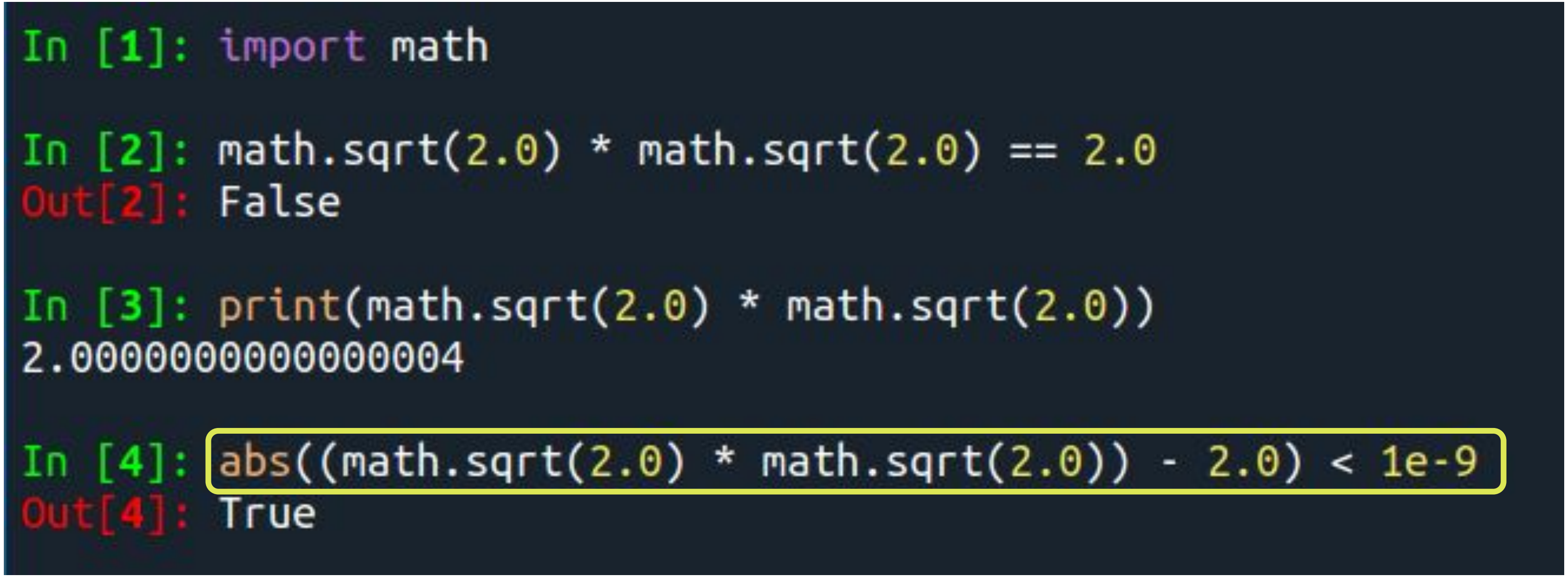
$$1.25 = 5 * 2^{-2} = (101, -10)$$

T/F Question!

T/F Question: The value of `math.sqrt(2.0) * math.sqrt(2.0) == 2.0` is `True`.

False! 2.0000000000000004 != 2.0

Approximation - Floats



```
In [1]: import math

In [2]: math.sqrt(2.0) * math.sqrt(2.0) == 2.0
Out[2]: False

In [3]: print(math.sqrt(2.0) * math.sqrt(2.0))
2.0000000000000004

In [4]: abs((math.sqrt(2.0) * math.sqrt(2.0)) - 2.0) < 1e-9
Out[4]: True
```

Swap Variables: Question

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

Swap Variables

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

❌

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

☺

Very pythonic!

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

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: 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

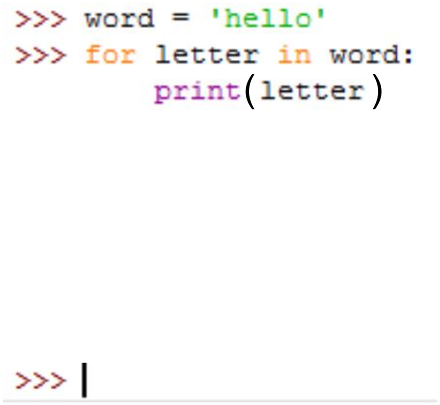
while

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

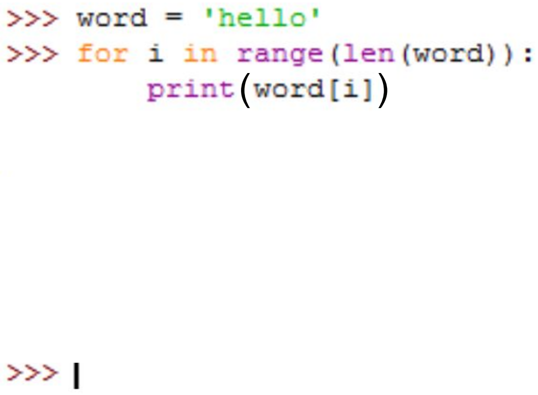
- while *condition*:

      #code

Control: For Loops

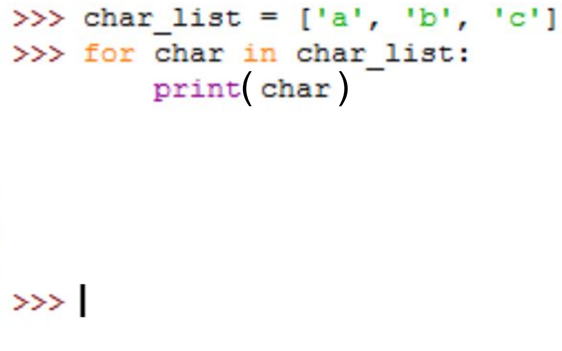


```
>>> word = 'hello'
>>> for letter in word:
        print(letter)

>>> |
```

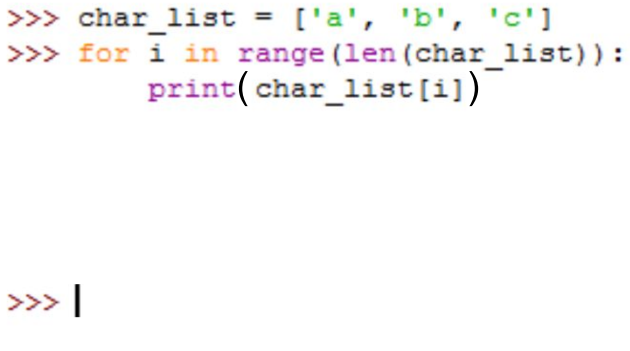


```
>>> word = 'hello'
>>> for i in range(len(word)):
        print(word[i])

>>> |
```

Think about if you want the element in the iterable itself, or just its index



```
>>> 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

What is going to be printed?

```
# what will be printed?

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)
```

Example Question

```python
# what will be printed?

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
```

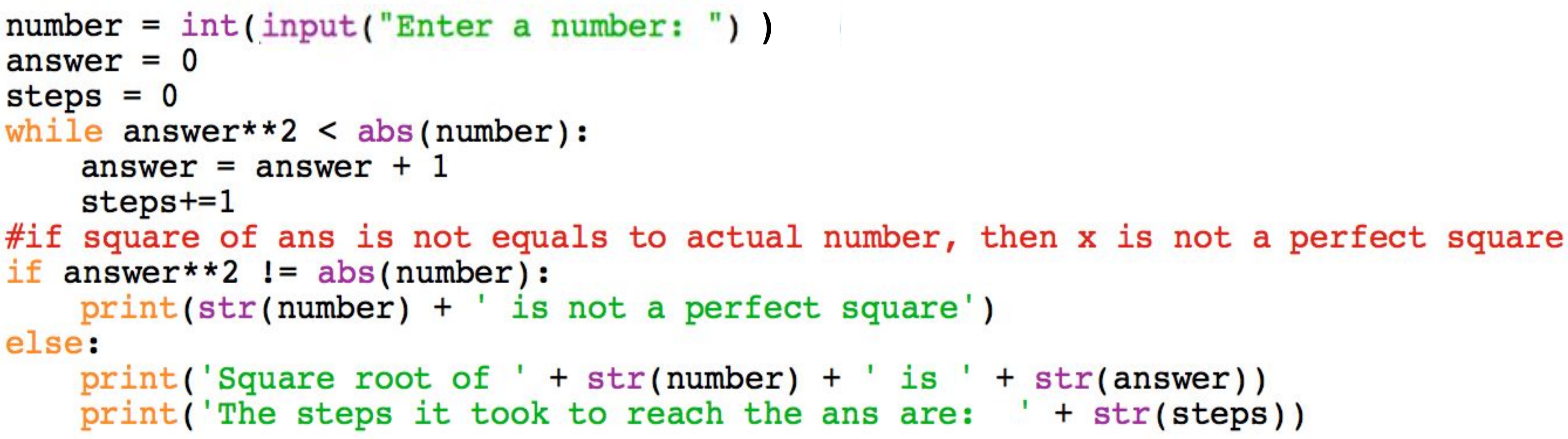
Last value of i was 1 → **1**

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



```
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

```
>>> 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 `int`s. In Python, the assignment statement `L[0][0] = 3` mutates the list `L`.

False

List Functions L = list('6.100A')

Don't iterate over a list you're modifying!

Function	Return value	L	Notes
len(L)	6	['6', '.', '1', '0', '0', 'A']	
L.append(['e'])	None	['6', '.', '1', '0', '0', 'A', ['e']]	
L.extend(['b', 'a'])	None	['6', '.', '1', '0', '0', 'A', 'b', 'a']	
L + ['b', 'a']	['6', '.', '1', '0', '0', 'A', 'b', 'a']	['6', '.', '1', '0', '0', 'A']	creates a new list
L.insert(2, 'c')	None	['6', '.', 'c', '1', '0', '0', 'A']	
L.remove('.')	None	['6', '1', '0', '0', 'A']	error if element not in L
L.reverse()	None	['A', '0', '0', '1', '.', '6']	
L.pop()	'A'	['6', '.', '1', '0', '0']	can take an optional index
L.sort()	None	['.', '0', '0', '1', '6', 'A']	can take an optional key parameter
sorted(L)	['.', '0', '0', '1', '6', 'A']	['6', '.', '1', '0', '0', 'A']	can take an optional key parameter
enumerate(L)	[(0, '6'), (1, '.'), (2, '1'), (3, '0'), (4, '0'), (5, 'A')]	['6', '.', '1', '0', '0', 'A']	

List Indexing

```
L[start:stop:step]
defaults to    0    len(L)    1
         inclusive  exclusive
```

```
>>> letters = ['a', 'b', 'c', 'd', 'e']
>>> letters[:]  shallow copy
['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]  reverse
['e', 'd', 'c', 'b', 'a']
>>> letters[1:4:2]
['b', 'd']
```

List Comprehensions

A fast, pythonic trick!

Create a **new** list using the values in an existing one:

```
L1 = [0, 4, 8, 16]
L2 = [x ** 2 for x in L1]
print(L1)
print(L2)
```

```
 >>> [0, 4, 8, 16]
 >>> [0, 16, 64, 256]
```

Tuples

Like lists, but immutable

```
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]`

```
(2,3)
```

- 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: True

In Python, the keys of a dictionary must be immutable.

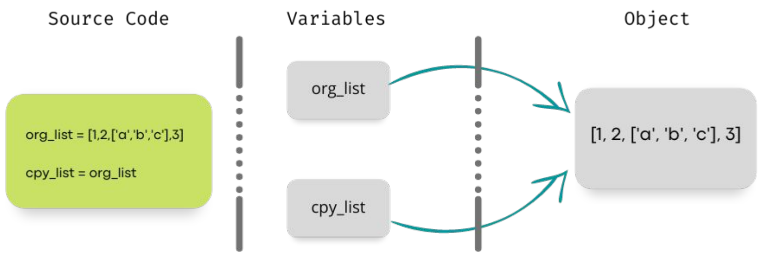
T/F Question: False

The dictionary `{'a':'1', 'b':'2', 'c': '3'}` has a mapping of `str` to `int`

Using Dictionaries

```
zoo = {'elephant': 3, 'giraffe': 4}
print(len(zoo))        # 2
print(zoo['elephant']) # 3

try:
    print(zoo['frog'])
except Exception as e:
    print(e) # KeyError: 'frog'

print(zoo.get('frog', 'Frog is not in zoo')) # Frog is not in zoo

if 'cheetah' not in zoo:
    zoo['cheetah'] = 5

print(zoo) # {'elephant': 3, 'giraffe': 4, 'cheetah': 5}
print(list(zoo.keys()))   # ['elephant', 'giraffe', 'cheetah']
print(list(zoo.values())) # [3, 4, 5]

del zoo['elephant']
print(zoo) # {'giraffe': 4, 'cheetah': 5}
```

Mutability & Aliasing



Source Code

Variables

Object

org_list = [1,2,['a','b','c'],3]

cpy_list = org_list

org_list

cpy_list

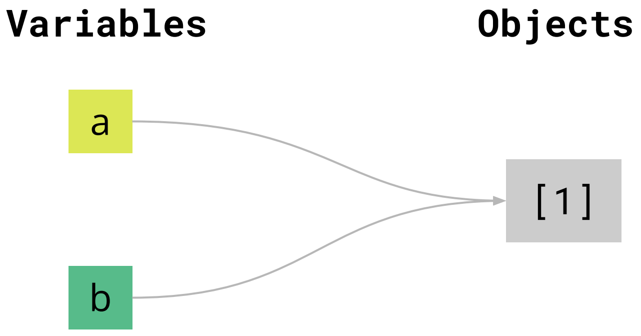
[1, 2, ['a', 'b', 'c'], 3]

Mutable: Lists, Dictionaries, Sets

Immutable: Strings, int, float, bool, tuples

Aliasing: Two variables bound to the same object

```
>>> a = [1]
>>> b = a
>>> a.append(2)
>>> print(a)
[1, 2]
>>> print(b)
[1, 2]
```



Variables

Objects

a

b

[1]

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, just know its **inputs** and **outputs**

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

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 ⇒ running it, with specific parameters

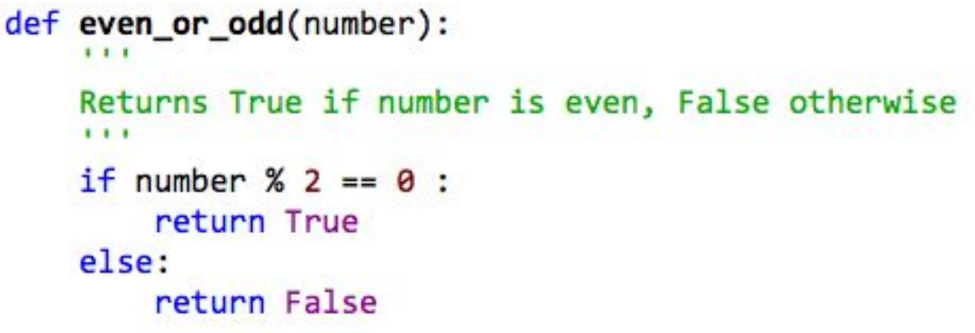
How to call a function:

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

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

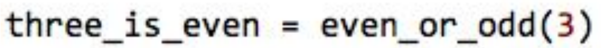
Functions examples

function definition



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

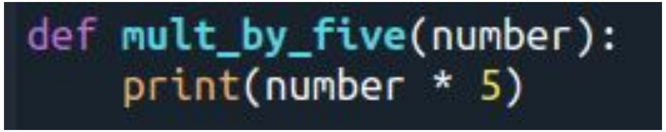
function call



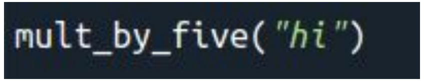
```
three_is_even = even_or_odd(3)
```

Question: what’s the difference between **even_or_odd()** and **even_or_odd** in code?

Functions examples



```
def mult_by_five(number):
    print(number * 5)
```



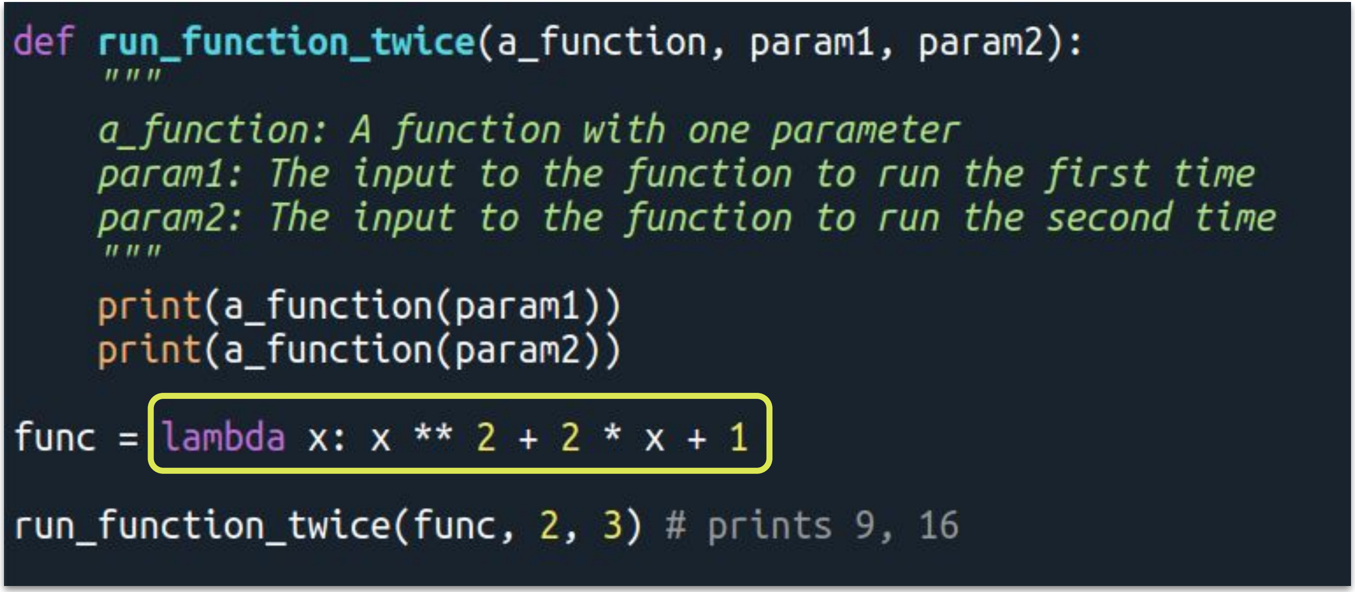
```
mult_by_five("hi")
```

what does this do? **<u>Prints: hihihihihi</u>**

what is returned by **mult_by_five** ? **<u>None !</u>**

Lambda functions

A way to define a short function in one line, often to be used as an argument to another function:



```python
def run_function_twice(a_function, param1, param2):
    """
    a_function: A function with one parameter
    param1: The input to the function to run the first time
    param2: The input to the function to run the second time
    """
    print(a_function(param1))
    print(a_function(param2))

func = lambda x: x ** 2 + 2 * x + 1

run_function_twice(func, 2, 3) # prints 9, 16
```

Scope

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

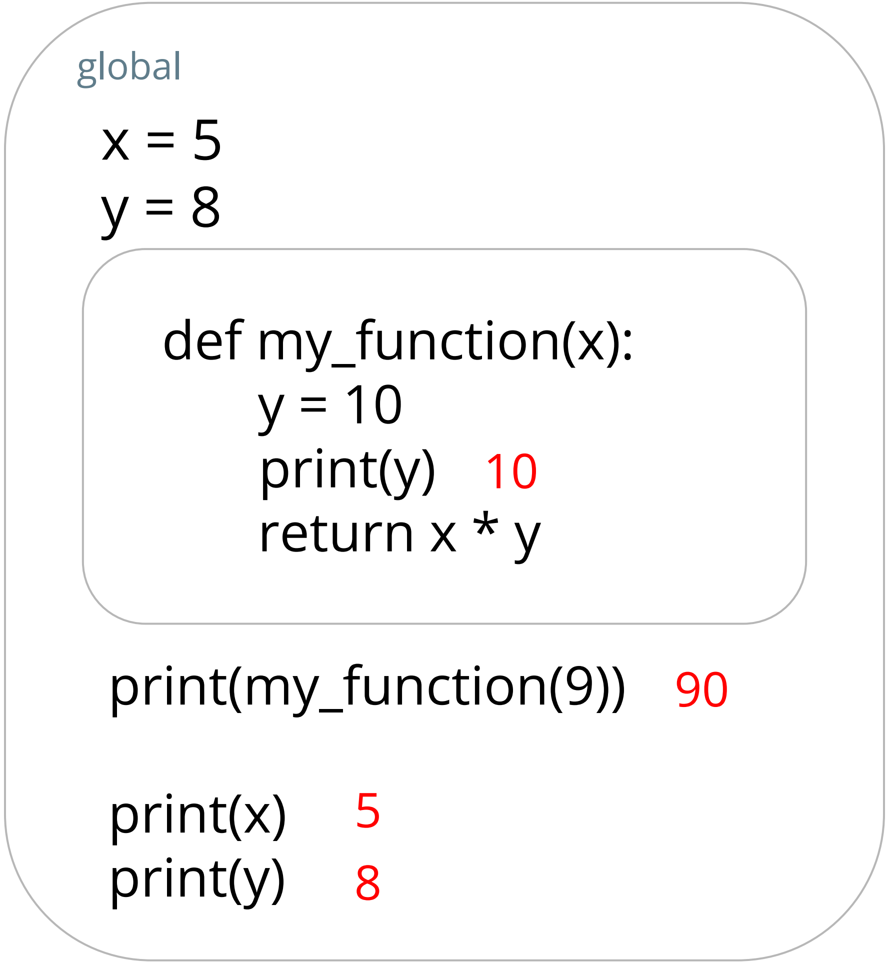
- a scope is a table, mapping variable names to values
  - assignment (<variable> = <expression>) adds an item to the table
- when your program starts, there's one scope called *global* scope
- when you call a function, a new scope is created
  - the scope is destroyed when the function returns

Scope

How is scope used?

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

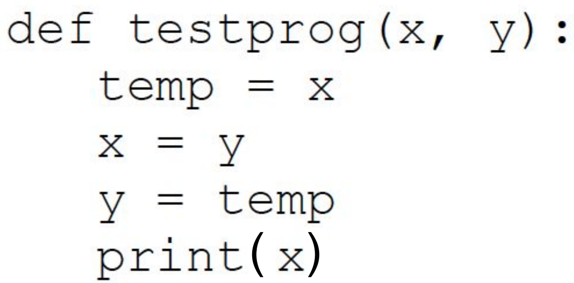
Scope



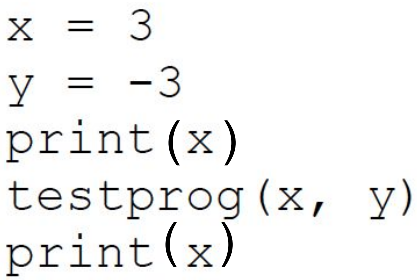
global

```
x = 5
y = 8

    def my_function(x):
        y = 10
        print(y)    10
        return x * y

print(my_function(9))   90

print(x)    5
print(y)    8
```

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?

3
-3
3

Scope

```
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)
```

```
in f (x) : x = 1
in f (x) : y = 2

in g () : x = 1
```

Recursion

a *recursive* function is any function that calls itself

Two crucial structural characteristics:

- **Base case**: any input that can be solved immediately
    - 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?

when a problem can be solved easily *if*

we have the answer to a <u>subproblem of the same form</u>

Recursion (example: Deep Copy)

given a potentially nested list of ints, generate a deep copy of the list (so that modifying any level of the copy does not modify the original nested list)

```
original = [1, [2, [3, [4]]]]
copy = deepcopy(original)

copy[1][1][1][0] = "MODIFIED"
print(original)
print(copy)
```

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

Recursion (example: Deep Copy)

Base case:

- If what we are copying doesn't require a deep copy, we can copy it immediately.
  - `int`s don't require us to do any special copy operation.

Recursive case:

- we need to make a deep copy of any nested `list`s to avoid aliasing

```python
def deepcopy(L):
    """
    Args:
        L: a potentially nested list of lists

    Returns:
        list: An exact copy of L (with nested structure)
    """

    ret = []

    for i in L:
        if type(i) is int:
            # base case
            ret.append(i)
        else:
            # recursive case
            ret.append(deepcopy(i))

    return ret
```

Recursion (example: Cartesian product)

given a list of lists, generate all combinations by selecting one item from each sublist (this is known as the *Cartesian product*)

```
print(cartesian_product([
     ["red shoes", "white shoes", "black shoes"],
     ["blue jeans", "gray jeans", "tan khakis"]
     ]))
```

```
[
     ('red shoes', 'blue jeans'), ('red shoes', 'gray jeans'), ('red shoes', 'tan khakis'),
     ('white shoes', 'blue jeans'), ('white shoes', 'gray jeans'), ('white shoes', 'tan khakis'),
     ('black shoes', 'blue jeans'), ('black shoes', 'gray jeans'), ('black shoes', 'tan khakis')
]
```

Recursion (example: Cartesian product)

Base case: only one category

-
  ```
  L = [["leather boots", "white sneakers", "black sneakers"]]
  return [(i,) for i in L[0]]
  ```

Recursive case:

- Given the combinations of the other categories (e.g. shirts and pants), we only need to add in the options for the current category (e.g. shoes) we are looking at

```python
def cartesian_product(L):
    """
    Args:
        L: a list of lists
    Returns:
        list: A list of tuples where each tuple is created
              by selecting one element from each sublist of L.
    """
    # base case
    if len(L) == 0:
        return []

    if len(L) == 1:
        options = L[-1] # equivalent to L[0]
                        # used L[-1] for consistency with recursive case
        return [(i,) for i in options]

    # recursive case
    existing_combos = cartesian_product(L[:-1])

    ret = []
    for combo in existing_combos:
        options = L[-1]
        for i in options:
            ret.append(combo + (i,))

    return ret
```

Recursion (additional problems)

- generate all permutations of the elements in L
- flatten nested lists

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? Θ? Ω?

O(g(n)**)**

Describes an **upper bound** on the runtime complexity

Ω(g(n)**)**

Describes a **lower bound** on the runtime complexity

Θ(g(n)**)**

Describes the **tight (upper and lower) bound** on the runtime complexity

Big Θ Notation

- Describes 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

Examples

I have a function $f(x) = 3x^2 + 2x + 1$

O(g(n)**)**

Can be $O(n^2)$, because there exists $g(n) = 4x^2$ that will surpass $f(x)$

But could also be $O(n^3)$, for example, since it will always surpass $f(x)$

Ω(g(n)**)**

Could be $\Omega(n)$, meaning the complexity will always surpass $g(n) = n$

Θ(g(n)**)**

Must be $\Theta(n^2)$, because it is both an upper and lower bound

i.e. $4x^2$ and x^2

Commonly Used Complexities in Algorithms

$\Theta(1)$ - Constant

$\Theta(\log n)$ - Logarithmic

$\Theta(n)$ - Linear

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

$\Theta(n^k)$ - Polynomial

$\Theta(k^n)$ - Exponential

Big Θ Notation Mechanics

Fastest growing term dominates:

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

Constant factors do not affect complexity:

$1000000000n$ $\qquad$ $0.0000001n$

Big Θ Notation Mechanics

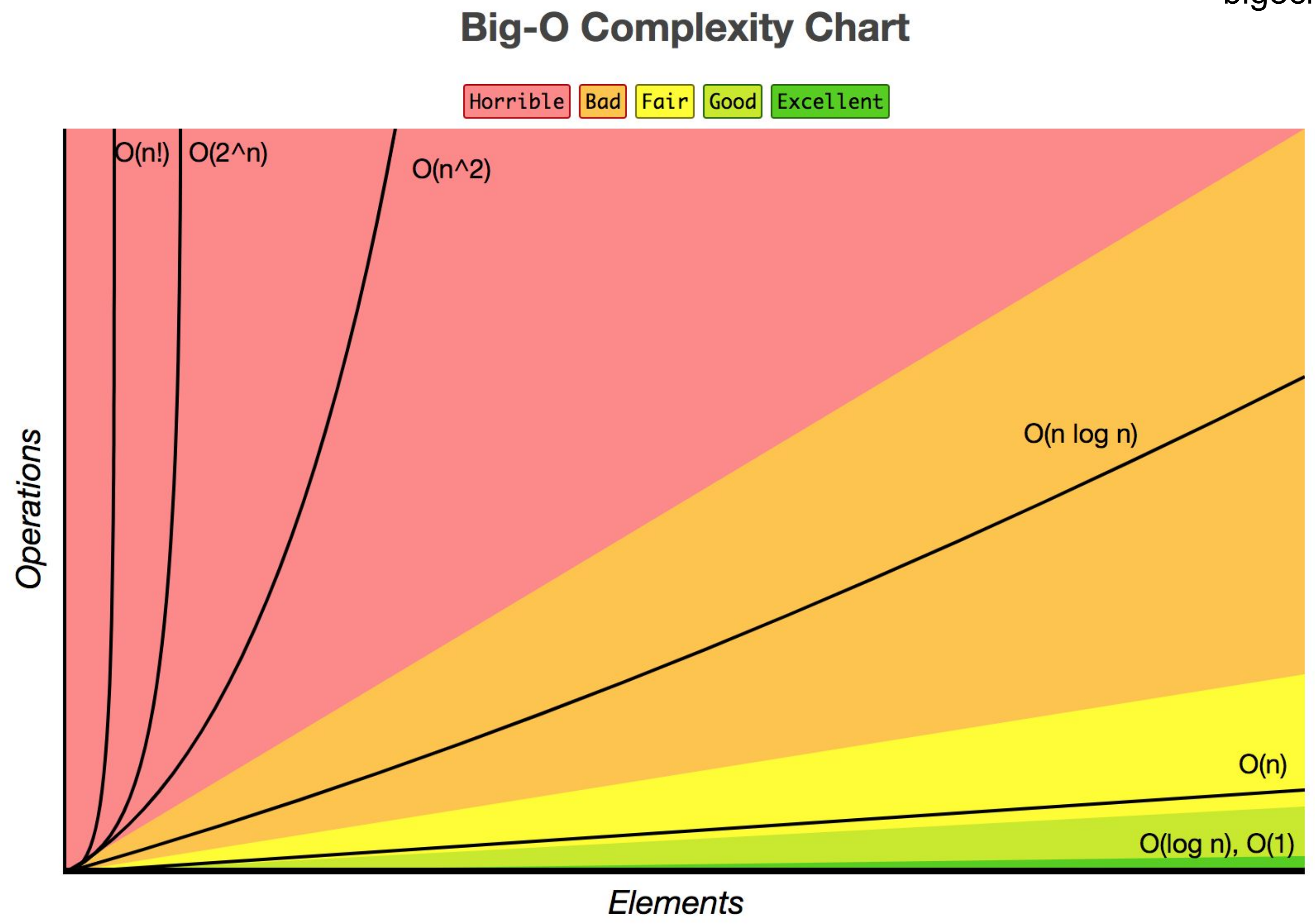
Fastest growing term dominates:

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

Constant factors do not affect complexity:

~~1000000000~~$n = \Theta(n) =$ ~~0.0000001~~n

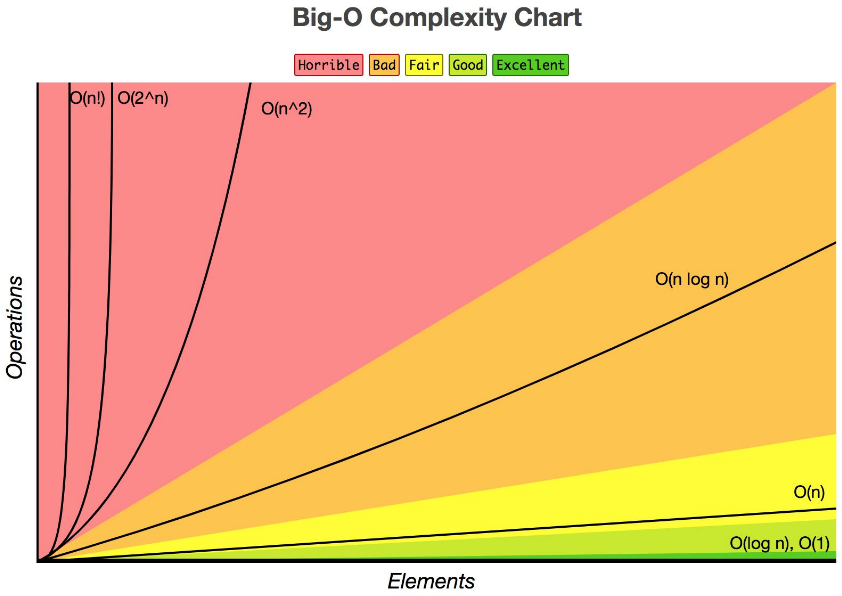
bigocheatsheet.com



Big Θ Notation Mechanics

Fastest growing term dominates:

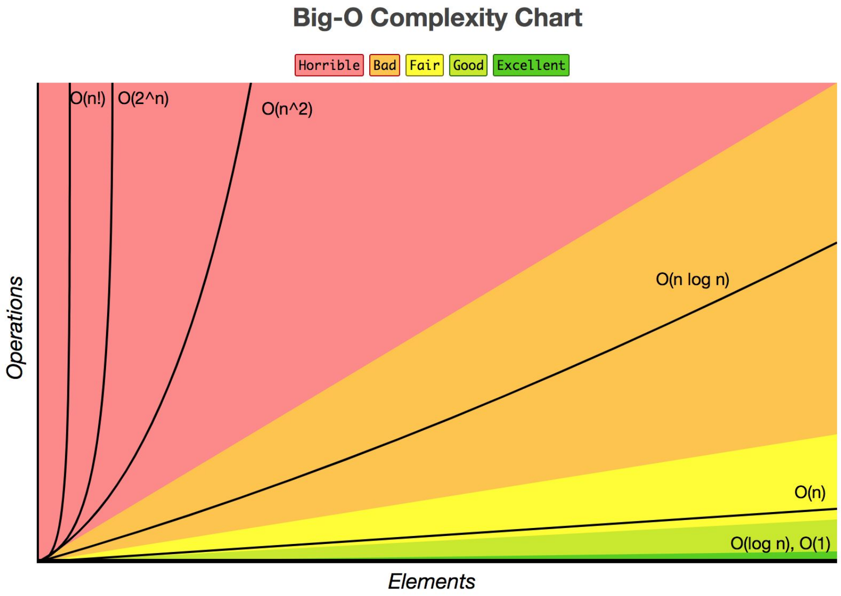
$n^2 + 100n + 1000 \log(n)$



Big Θ Notation Mechanics

Fastest growing term dominates:

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



Complexity of built-in methods

- **Constant-time, Θ(1)**
  - Assignment, `x = 2`
  - Basic operations, `+ - * / > <`
- **Dictionary**
  - Look-up: Θ(1)
  - Length: Θ(1)
  - Insert: Θ(1)
  - Delete: Θ(1)
  - `dictionary.keys()`: Θ(n) - because a list is generated
  - Check if a key is in the dictionary: Θ(1)

Complexity of built-in methods

- **List**
  - Append: Θ(1)
  - Length: Θ(1)
  - Insert: Θ(n)
  - Delete: Θ(n)
  - Copy: Θ(n)
  - Sort (`L.sort()`, `sorted()`): Θ(n log n)
  - Check if an item is in the list: Θ(n)
    - `if elt in L:`

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: Θ(log n)

What is the complexity?

```python
def is_palindrome_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: Θ(n)
Number of operations in each iteration: constant
Complexity: Θ(n)

What is the complexity?

```
def is_palindrome_recursive(s):
    if len(s) in {0, 1}:
        return True

    first_char = s[0]
    last_char = s[-1]

    if first_char != last_char:
        return False

    return is_palindrome_recursive(s[1:-1])
```

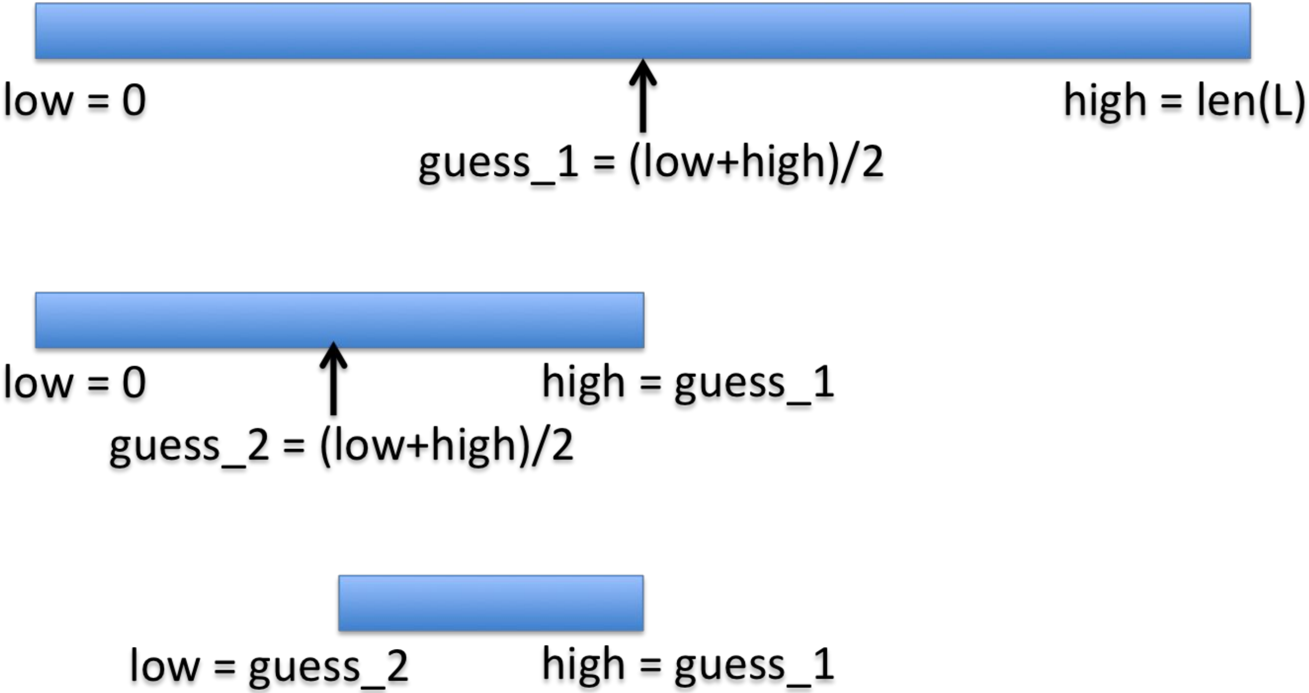
n / 2 recursive calls: Θ(n)
Slicing strings: Θ(n)
Complexity: Θ(n^2)

Slicing a string = Θ(n)

Search

- Linear search
  - Brute force search
  - List doesn’t have to be sorted
  - Θ(n)
- Bisection search
  - List must be sorted to give correct answer
  - Θ(log n)

Bisection search



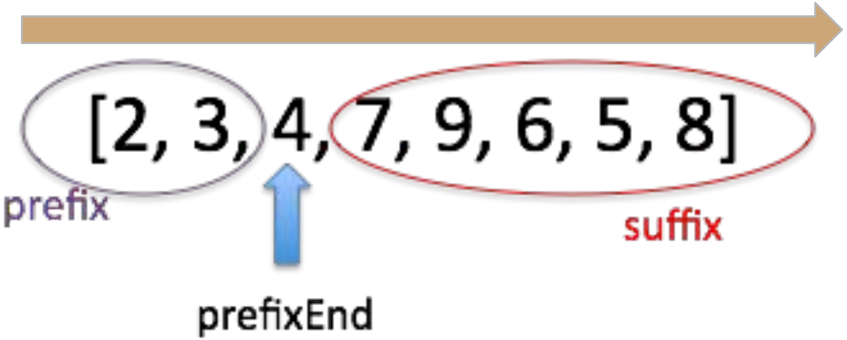
Complexity of searching unsorted list

- Linear search
    - Θ(n)
    - One time search

- Bisection search
    - complexity(sort) + complexity(bisection search)
    - complexity(sort) + Θ(log n)
    - complexity(sort) > Θ(n), always* *(in 6.006 you'll learn a caveat to this)*

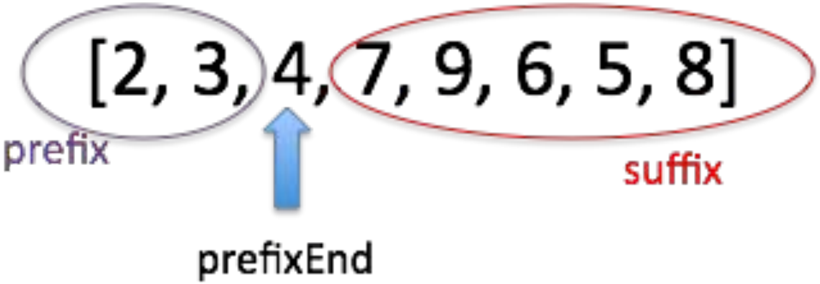
Sorting Methods: Selection Sort

- 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



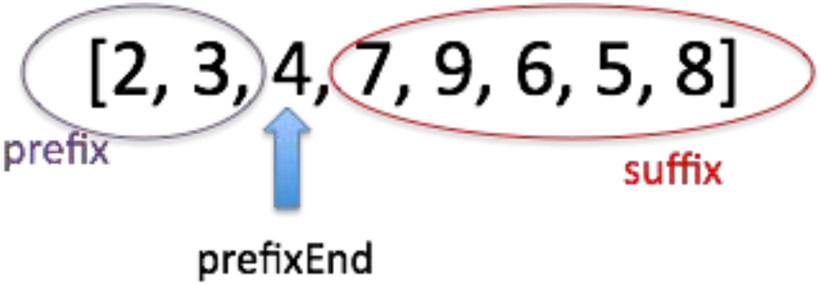
Sorting Methods: Selection Sort

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



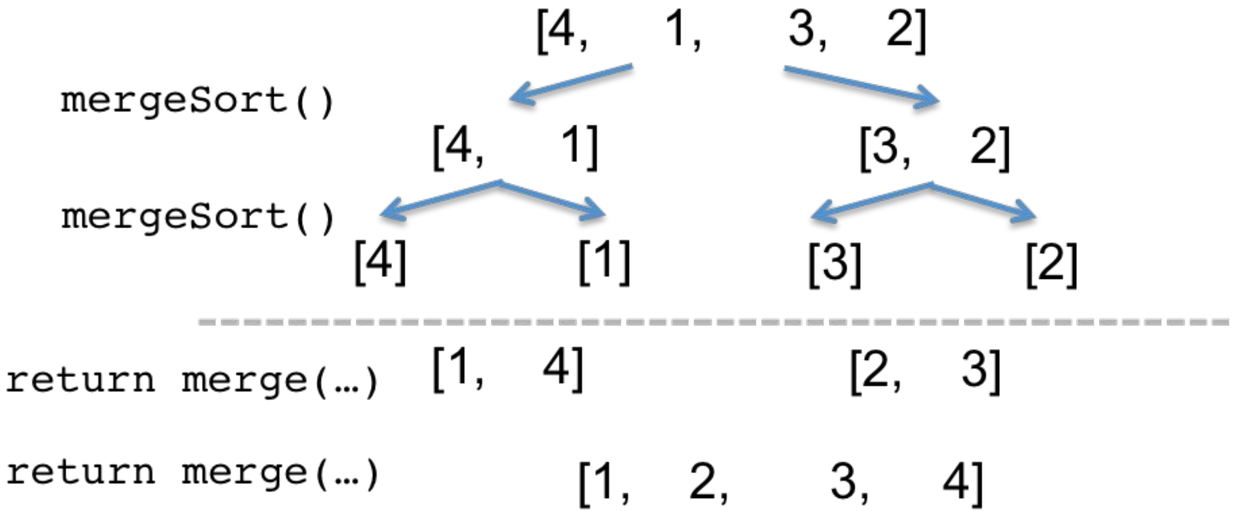
Sorting Methods: Selection Sort

- How many steps? Θ(n)
- Each step, how many operations? 1, 2, 3, ..., n
- Complexity? Θ(n^2)



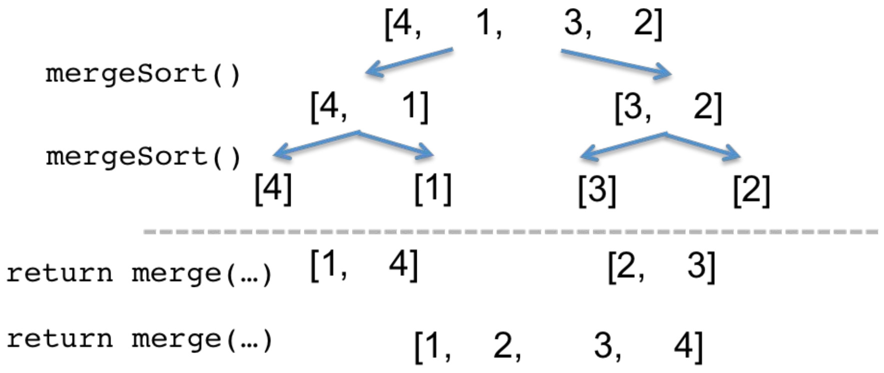
Sorting Methods: Merge Sort

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



Sorting Methods: Merge Sort

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



[4, 1, 3, 2]

mergeSort()

[4, 1] [3, 2]

mergeSort()

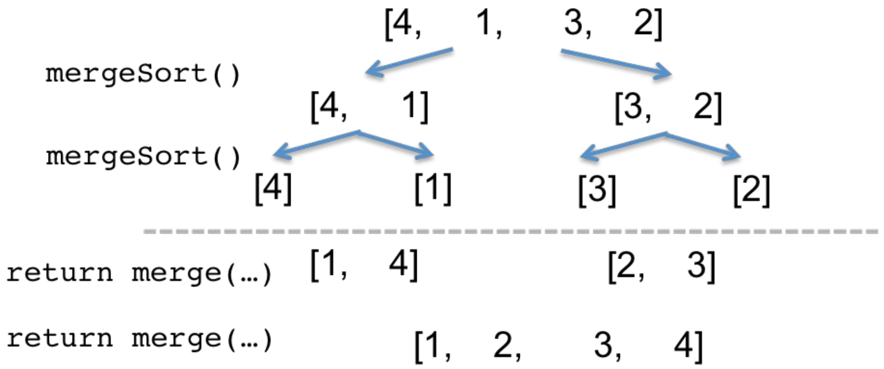
[4] [1] [3] [2]

return merge(…) [1, 4] [2, 3]

return merge(…) [1, 2, 3, 4]

Sorting Methods: Merge Sort

- How many levels of the recursive tree? Θ(log n)
- How much computation of each level of the tree? Θ(n)
- Complexity? Θ(n log n)



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>
```

Assertion Error

```
x = 3
assert x == 4, 'x is not 4’
```

throws an `AssertionError` and stops all further computation

Exception Types

- `NameError`: e.g. access a name to a variable that doesn't exist
  - ex. NameError: name 'variable_name' is not defined
- `ValueError`: e.g. concatenating a non-string with a string
- `IndexError`: e.g. accessing beyond the limits of a list
  - ex. IndexError: list index out of range
- `KeyError`: e.g. 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`: e.g. trying to append to a string
  - ex. AttributeError: 'str' object has no attribute 'append'

OBJECT ORIENTED PROGRAMMING

Classes

Classes provide a means of bundling data and functionality together

They have (instance) **attributes** and (instance) **methods** specific to themselves

```
class Vehicle(object):
    def __init__(self, name):
        # an instance attribute
        self.name = name

    def get_name(self):
        return self.name

    def honk(self):
        # an instance method
        print(f"{self.name} says HONK")
```

You can instantiate **objects** from classes

```
my_vehicle = Vehicle("batmobile")
print(my_vehicle.get_name())   # prints: batmobile
my_vehicle.honk()              # prints: batmobile says HONK
```

```
class Vehicle(object):
    def __init__(self, name):
        # an instance attribute
        self.name = name

    def get_name(self):
        return self.name

    def honk(self):
        # an instance method
        print(f"{self.name} says HONK")
```

Instance methods belong to the **object** itself

```
my_vehicle = Vehicle("batmobile")
print(my_vehicle.get_name())   # prints: batmobile
my_vehicle.honk()              # prints: batmobile says HONK
print(my_vehicle.honk)
# prints:
# <bound method Vehicle.honk of <__main__.Vehicle instance at
0x1010748c0>>
```

```
class Vehicle(object):
    def __init__(self, name):
        # an instance attribute
        self.name = name

    def get_name(self):
        return self.name

    def honk(self):
        # an instance method
        print(f"{self.name} says HONK")
```

Inheritance

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

- Vehicle is the **parent class (superclass)**
- Car is the **child class (subclass)**
- Car.honk exists, even if we did not explicitly define it!
- Inheritance helps us enforce the **substitution principle**: Behaviors of the supertype should be supported by each of its subtypes.
- Does Vehicle.beep exist? No!

```python
class Vehicle(object):
    def __init__(self, name):
        # an instance attribute
        self.name = name

    def get_name(self):
        return self.name

    def honk(self):
        # an instance method
        print(f"{self.name} says HONK")
```

```python
class Car(Vehicle):
    def __init__(self, name):
        super().__init__(name)
        self.type = "car"

    def beep(self):
        print(f"{self.name} says BEEP")
```

We use super() to inherit attributes and methods from the parent class.

```
class Vehicle(object):
    def __init__(self, name):
        # an instance attribute
        self.name = name

    def get_name(self):
        return self.name

    def honk(self):
        # an instance method
        print(f"{self.name} says HONK")
```

```
class Car(Vehicle):
    def __init__(self, name):
        super().__init__(name)
        self.type = "car"

    def beep(self):
        print(f"{self.name} says BEEP")
```

We use super() to find the parent class and call its **__init__** function

Polymorphism lets us override inherited functions

We can tailor what inherited functions do for our specific subclass!

```
vehicles = [Truck("matt"),
Vehicle("ashhad"), Car("esha")]

for v in vehicles:
    v.honk()
```

```
class Vehicle(object):
    def __init__(self, name):
        # an instance attribute
        self.name = name

    def get_name(self):
        return self.name

    def honk(self):
        # an instance method
        print(f"{self.name} says HONK")
```

```
class Car(Vehicle):
    def __init__(self, name):
        super().__init__(name)
        self.type = "car"

    def honk(self):
        print(f"{self.name} says BEEP")
```

```
class Truck(Vehicle):
    def __init__(self, name):
        super().__init__(name)
        self.type = "truck"

    def honk(self):
        print(f"{self.name} says BIGHONK")
```