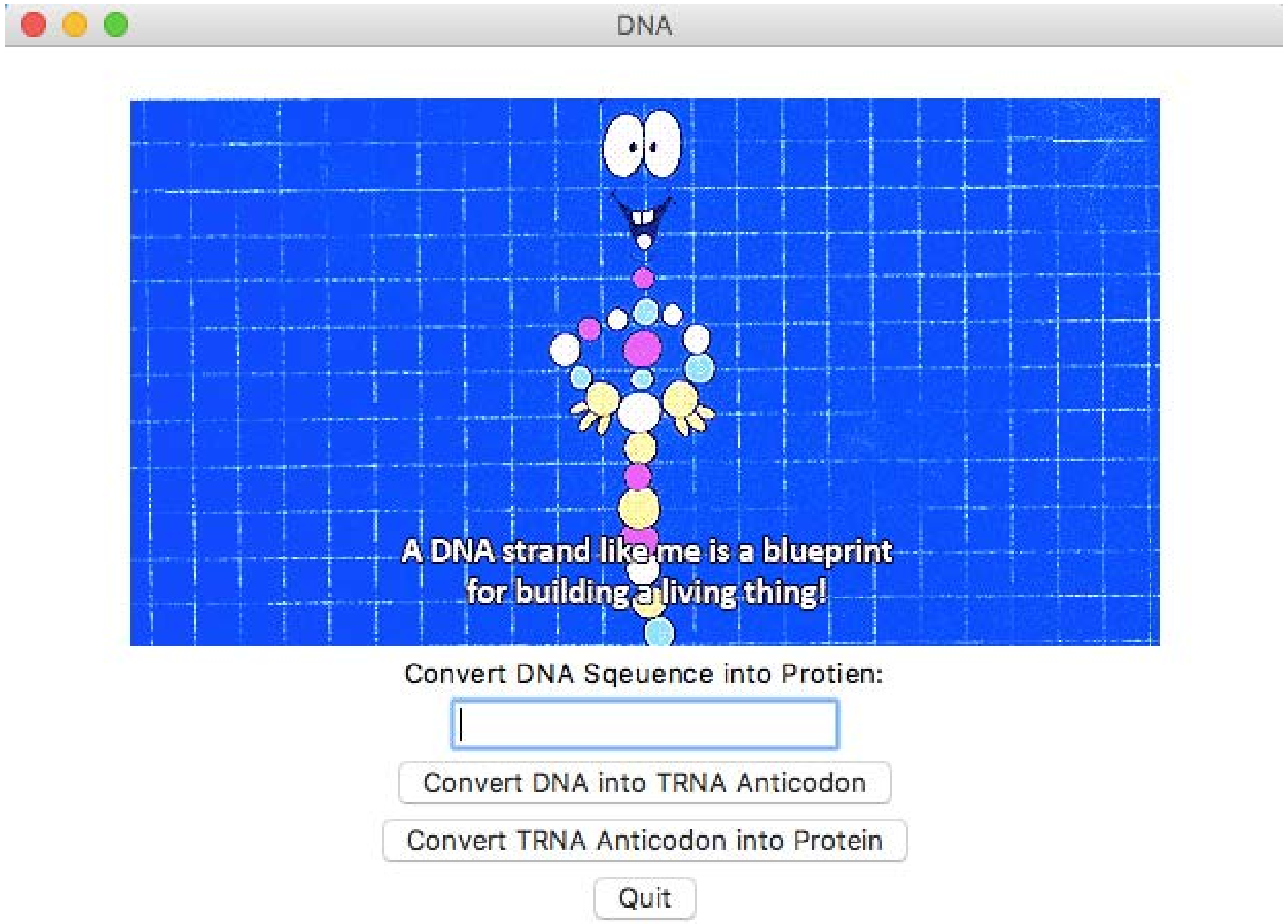
# DATA 520
# Lecture 18

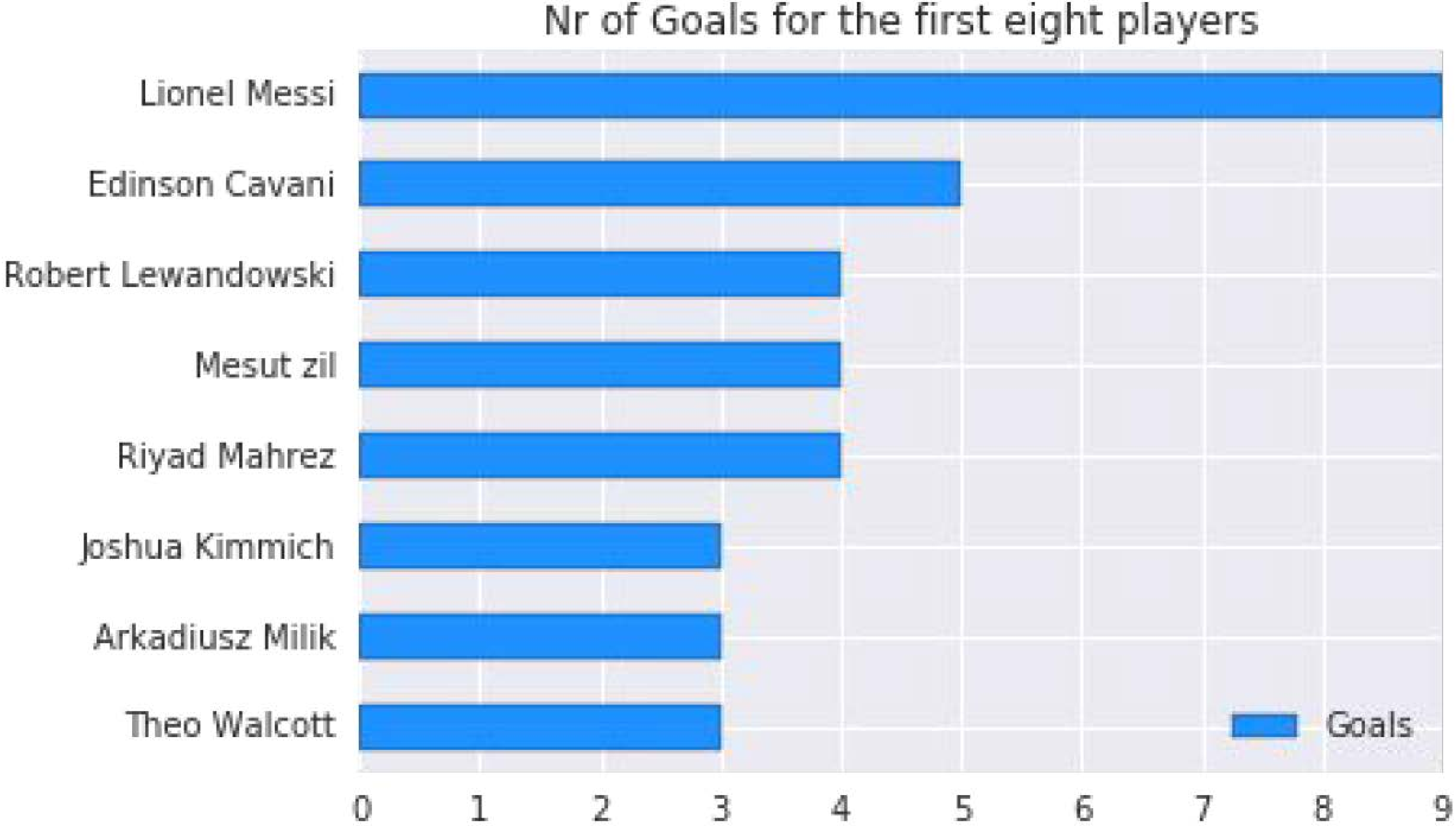
## Projects

## Searching Algorithms

# Translate DNA GUI

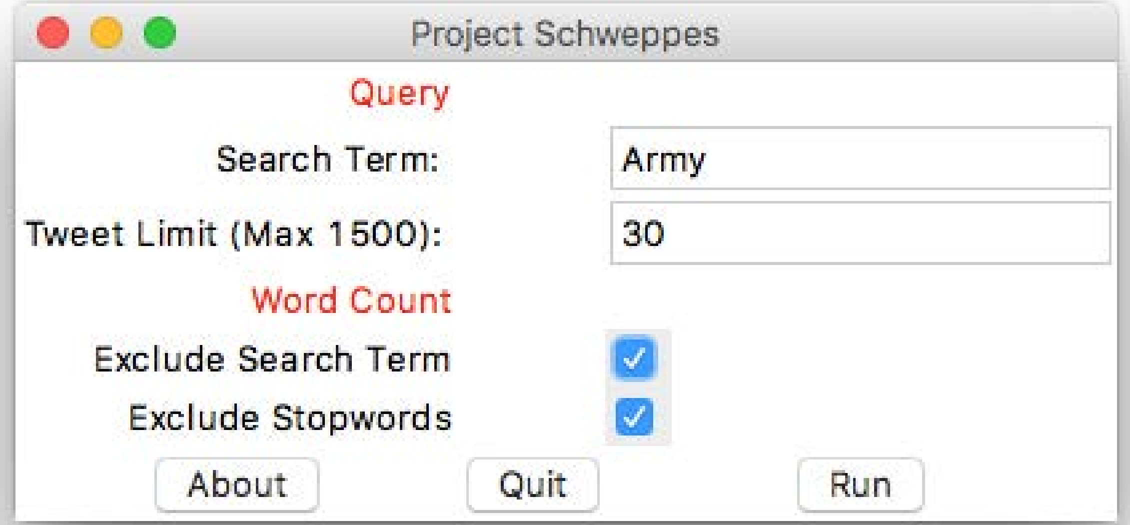


# web scraping with special modules



# SchweppesGUI.py

- Simple yet refined



# Monitoring program activity

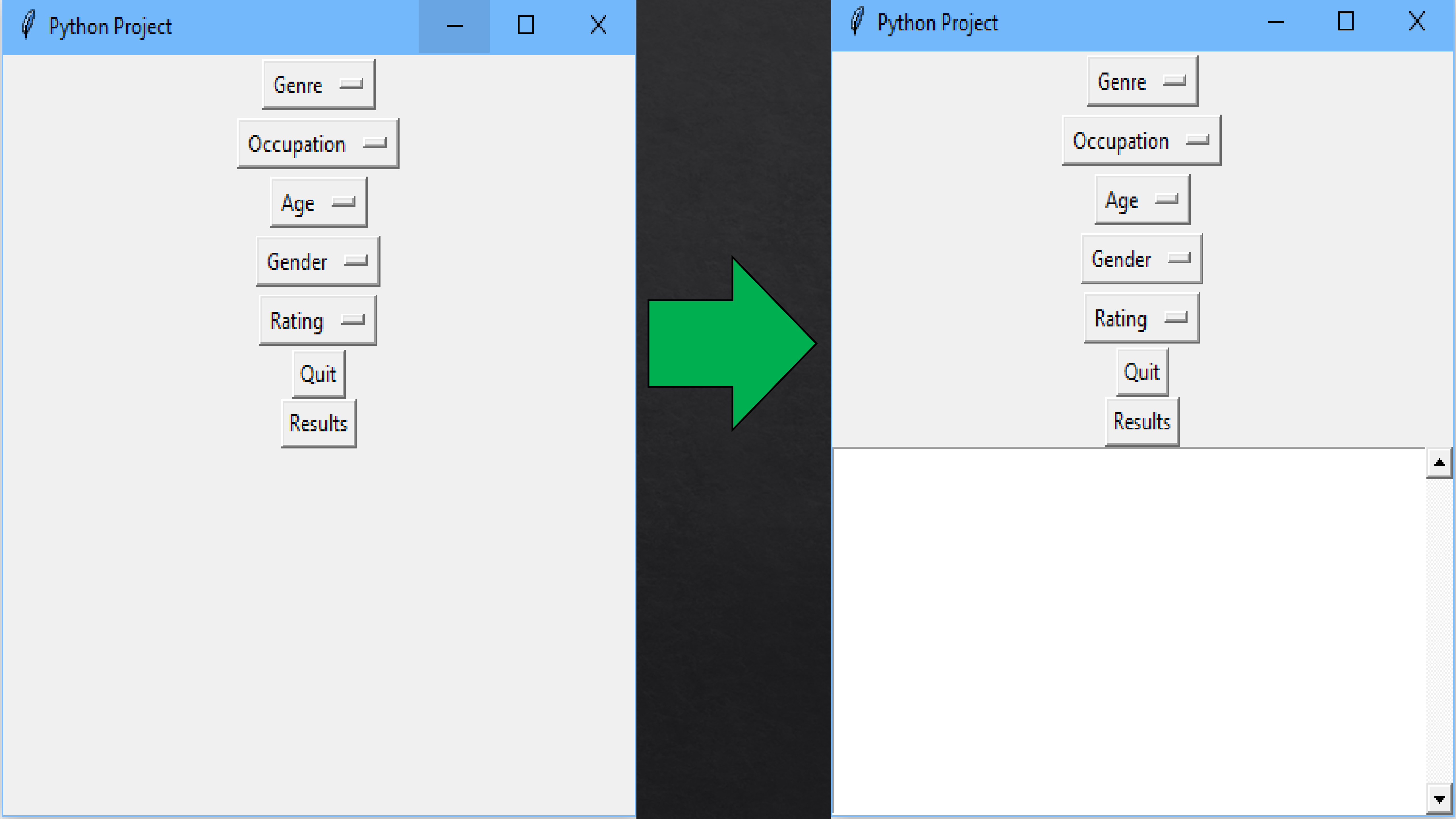
Runs a while loop that continuously runs the ID_Task function until the response is 'False'.

Commands the computer to start a shutdown timer if response is 'False' through the os module.

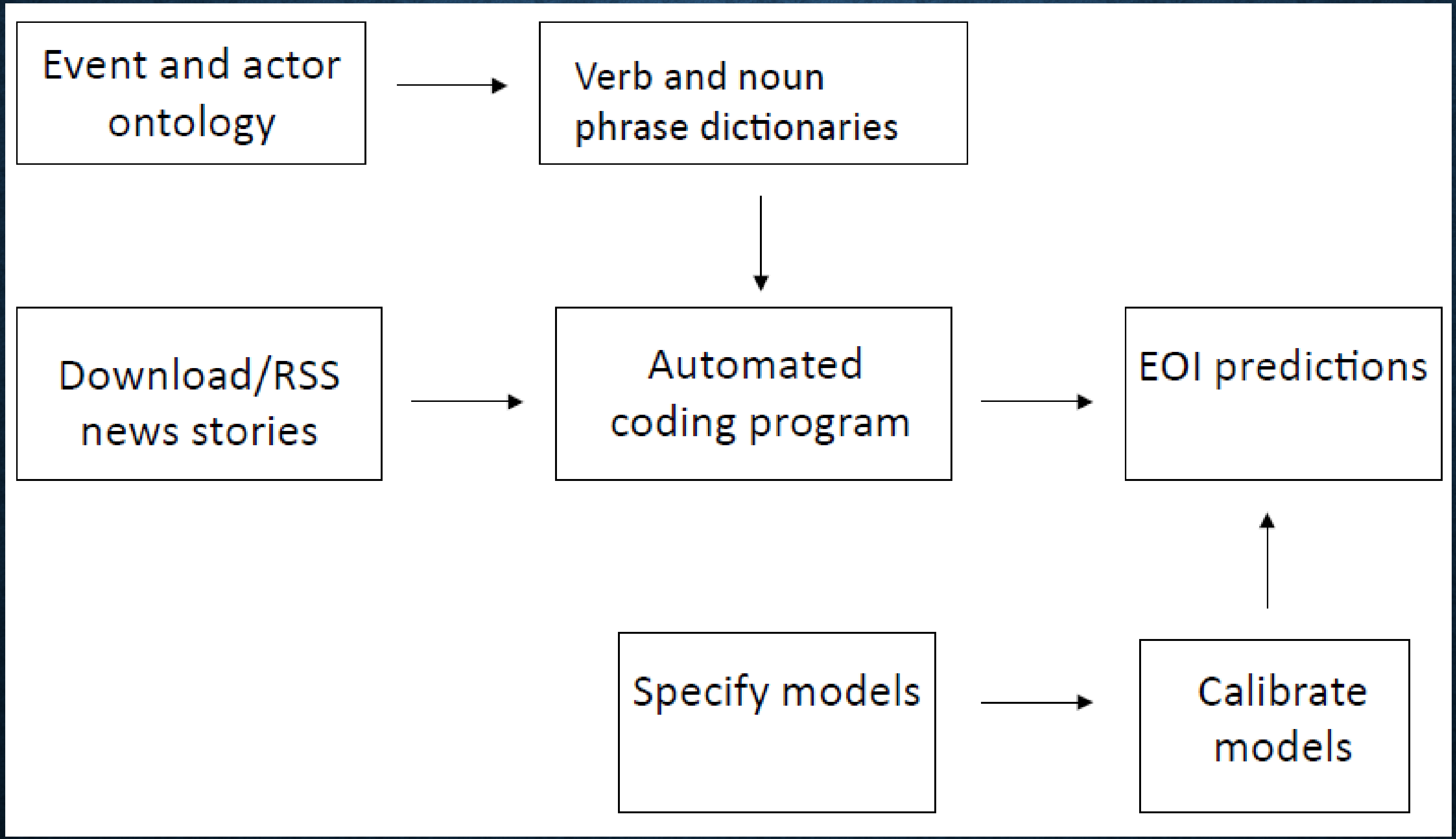
Asks to user to input what task they want to monitor and then runs the Check_process function.

```
def Check_process(task):
    ''' Check_process(arg) -> Check_process(str)
    Function runs the ID_Task function with the script continuously until the
    'task' is no longer running, at which point it shuts down the computer.
    >>> Check_process('chrome')
    chrome is still running
    chrome is still running
    [chrome shuts down and shutdown process is initialized]
    '''
    while ID_Task(task) is True:
        #print ("{} is still running".format(task))
        pass
    else:
        #print("os.system('shutdown.exe -s -t 120')")
        os.system('shutdown.exe -s -t 120')

task = input ("what process do you want to monitor? ")
Check_process(task)
```


Python Project
Genre
Occupation
Age
Gender
Rating
Quit
Results
Python Project
Genre
Occupation
Age
Gender
Rating
Quit
Results

# PYTHON ENGINE FOR TEXT RESOLUTION AND RELATED CODING HIERARCHY(PETRARCH)



# Hockeytrack5

- Currently on version 5 hence the name
- Allows a person to rack shots and goals in realtime
  - Goals shown in blue
  - Shots in green
  - Total Shot and Goal counter
- Demo

# Searching

## A common thing that computers do

- more than one way to sort
- more than one way to search

Python uses a fast method: `L.index(value)`

**- returns index**

```
['d', 'a', 'b', 'a'].index('a')
1
```

How would we humans search? Look at each, one by one. Linear/sequential search

Python can do that too

# Searching

## Python can do a linear search

pseudocode:

**Look at each item in list. If it equals the value you are looking for, stop.**

```
def linear_search(vlist, srchval):  # somewhat different from book
    """ (list, object) -> int
    Return the index of the first occurrence of value in lst, or return
    -1 if value is not in lst.
    >>> linear_search([2, 5, 1, -3], 5)
    1
    >>> linear_search([2, 4, 2], 2)
    0
    >>> linear_search([2, 5, 1, -3], 4)
    -1
    >>> linear_search([], 5)
    -1
    """
```

# Searching

## Python can do a linear search

pseudocode:

```
Look at each item in list. If it equals the value you are looking for, stop.
# linear_search_1.py
# let's try a while statement first
# we need the index, and then a way to stop
index = 0 # index of list item examined

# vlist = values list; searchval
while index != len(vlist) and vlist[index] != srchval:
    index += 1

if index == len(vlist):
    return -1
else:
    return index
```

# Searching

## Python can do a linear search

pseudocode:

```
Look at each item in list. If it equals the value you are looking for, stop.
# linear_search_1.py
# let's try a while statement first
# we need the index, and then a way to stop
index = 0 # index of list item examined

# vlist = values list; searchval
while index != len(vlist) and vlist[index] != srchval:
    index += 1

if index == len(vlist):
    return -1
else:
    return index
```

two booleans

almost always true

# Searching

## Python can do a linear search another way: *for* instead of *while*

pseudocode:

```
look at each item in list. If it equals the value you are looking for, stop.
```

```
# linear_search_2.py
for item in vlist:
    if item == srchval:
        return index  # implicit break

return -1
```

**- looks more efficient!**

**This "for - if" is faster than the "while" because the length of vlist is set**

# Searching

## Python can also do a sentinel search

We will add the value at the end, guaranteeing we will find it. It will stop at the FIRST match anyway.

```
# linear_search_3.py
# add sentinel
vlist.append(srchval)

index = 0 # index of list item examined
while vlist[index] != srchval:
    index += 1

# remove sentinel
vlist.pop() # we have to remove it, the last one. we can use pop()

if index == len(vlist):
    return -1
else:
    return index
```

**- looks even more efficient!**

**Only one boolean statement needed**

# Timing the Searches

```
# time_searches.py part 1
import time
import linear_search_1
import linear_search_2
import linear_search_3
def time_it(search, L, v):
    """ (function, object, list) -> number
    Time how long it takes to run function search to find
    value v in list L.
    """
    t1 = time.perf_counter()
    search(L, v)
    t2 = time.perf_counter()
    return (t2 - t1) * 1000.0

def print_times(v, L):
    """ (object, list) -> NoneType
    Print the number of milliseconds it takes for linear_search(v, L)
    to run for list.index, the while loop linear search, the for loop
    linear search, and sentinel search.
    """
```

# Timing the Searches

```
# time_searches.py part 2 -  original from text - we will modify

    # Get list.index's running time.
    t1 = time.perf_counter()
    L.index(v)
    t2 = time.perf_counter()
    index_time = (t2 - t1) * 1000.0

    # Get the other three running times.
    while_time = time_it(linear_search_1.linear_search, L, v)
    for_time = time_it(linear_search_2.linear_search, L, v)
    sentinel_time = time_it(linear_search_3.linear_search, L, v)


    print("{0}\t{1:.2f}\t{2:.2f}\t{3:.2f}\t{4:.2f}".format(
        v, while_time, for_time, sentinel_time, index_time))

L = list(range(10000001)) # A list with just over ten million values

print_times(10, L) # How fast is it to search near the beginning?
print_times(5000000, L) # How fast is it to search near the middle?
print_times(10000000, L) # How fast is it to search near the end?
```

# Timing the Searches

```
# time_searches.py part 2 - modified output

    # Get list.index's running time.
    t1 = time.perf_counter()
    L.index(v)
    t2 = time.perf_counter()
    index_time = (t2 - t1) * 1000.0

    # Get the other three running times.

    while_time = time_it(linear_search_1.linear_search, L, v)

    for_time = time_it(linear_search_2.linear_search, L, v)

    sentinel_time = time_it(linear_search_3.linear_search, L, v)

    print("{0}\t{1:>6.1f}\t{2:>6.1f}\t{3:>6.1f}\t{4:>6.1f}".format(
        v, while_time, for_time, sentinel_time, index_time))


# modified to make it easier to change values

ListLength = 10000001 # default: 10,000,001

L = list(range(ListLength)) # A list of variable length

print  ('Search times in a list of ' , "{:,}".format(ListLength) ) # title,list size,commas
print ('Index\t while\t   for\t  Sent\t.index ')  # tabbed column headings
print_times(10, L) # How fast is it to search near the beginning?
print_times(round(ListLength/2), L) # How fast is it to search near the middle?
print_times(ListLength - 50, L) # How fast is it to search near the end?
```

# Timing the Searches

```
# time_searches.py
import time
import linear_search_1
import linear_search_2
import linear_search_3
def time_it(search, L, v):
    """ (function, object, list) -> number
    Time how long it takes to run function search to find
    value v in list L.
    """
    t1 = time.perf_counter()
    search(L, v)
    t2 = time.perf_counter()
    return (t2 - t1) * 1000.0

def print_times(v, L):
    """ (object, list) -> NoneType
    Print the number of milliseconds it takes for linear_search(v, L)
    to run for list.index, the while loop linear search, the for loop
    linear search, and sentinel search.
    """
# Get list.index's running time.
    t1 = time.perf_counter()
    L.index(v)
    t2 = time.perf_counter()
    index_time = (t2 - t1) * 1000.0

    # Get the other three running times.
    while_time = time_it(linear_search_1.linear_search, L, v)
    for_time = time_it(linear_search_2.linear_search, L, v)
    sentinel_time = time_it(linear_search_3.linear_search, L, v)


    print("{0}\t{1:>6.1f}\t{2:>6.1f}\t{3:>6.1f}\t{4:>6.1f}".format(
        v, while_time, for_time, sentinel_time, index_time))

# modified to make it easier to change values
ListLength = 10000001  # default: 10,000,001

L = list(range(ListLength)) # A list of variable length

print  ('Search times in a list of ' , "{:,}".format(ListLength) ) # title,list size,commas
print ('Index\t while\t   for\t  Sent\t.index ')  # tabbed column headings
print_times(10, L) # How fast is it to search near the beginning?
print_times(round(ListLength/2), L) # How fast is it to search near the middle?
print_times(ListLength - 50, L) # How fast is it to search near the end?
```

# Timing the Searches

## Results

Courier 16 point

```
Search times in a list of  1,000,001
Index   while    for   Sent .index
10        0.00   0.01   0.00   0.00
500000 119.72  46.47  66.73  10.10
999951 240.74  93.42 136.10  20.72
```

Courier 10 point

```
Search times in a list of  1,000,001
Index          while          for          Sent        .index
10              0.00         0.01          0.00          0.00
500000        119.72        46.47         66.73         10.10
999951        240.74        93.42        136.10         20.72
```

Lucida Console:

```
Search times in a list of  1,000,001
Index   while    for   Sent .index
10        0.00   0.01   0.00   0.00
500000 119.72  46.47  66.73  10.10
999951 240.74  93.42 136.10  20.72
```

```
Search times in a list of  10,000,001
Index      while      for      Sent   .index
10           0.00     0.01     0.00     0.00
5000000  1194.55   470.43   665.58   102.57
9999951  2410.80   947.65  1332.29   204.96
```

# Searching

**Is there a faster way to search a list?**

If the list is sorted!

How would you find "Sam Turner" in the phonebook?

- you use a binary search (before "Tu": not there yet, after: go back)

always bifurcating, diving by 2 at each step

1 step: 2; 2 steps: 4; 4 steps: 8;  formula = $2^{steps}$

so N values can be searched in $\log_2(N)$ steps;

get base 2 value of N

# Searching

**log base 2:**

| Searching *N* Items | Worst Case—Linear Search | Worst Case—Binary Search | $2^{steps}$ |
|---|---|---|---|
| 100 | 100 | 7 | 128 |
| 1000 | 1000 | 10 | 1024 |
| 10,000 | 10,000 | 14 | 16,384 |
| 100,000 | 100,000 | 17 | 131,072 |
| 1,000,000 | 1,000,000 | 20 | 1,048,576 |
| 10,000,000 | 10,000,000 | 24 | 16,777,216 |

**Table 18—Logarithmic Growth**

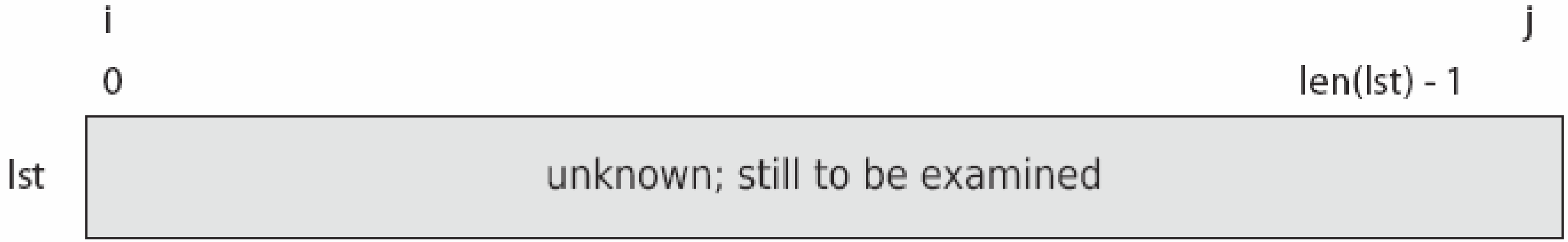
# Searching

## Searching in a sorted list:

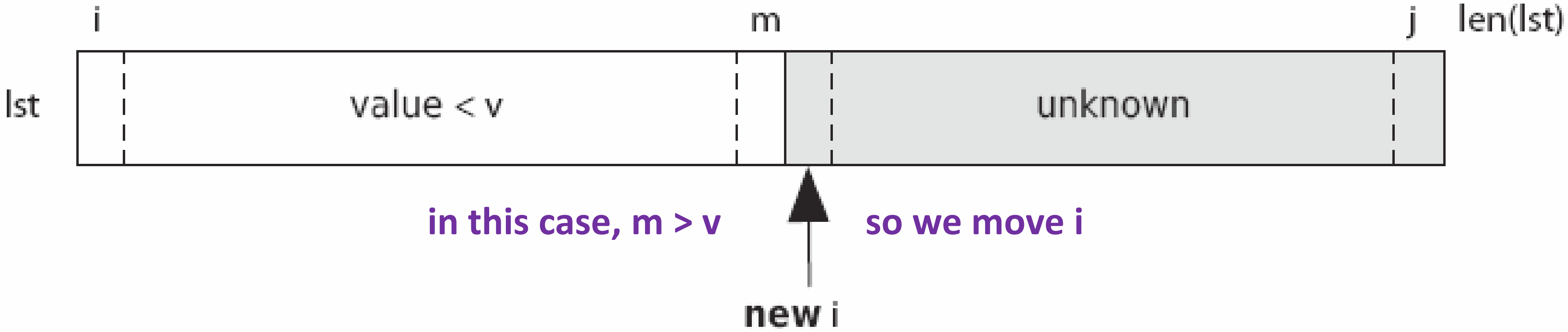
i = index of first item = 0 at beginning

j = index of last item to be examined = len(list)-1 at beginning

We are always searching from i to j



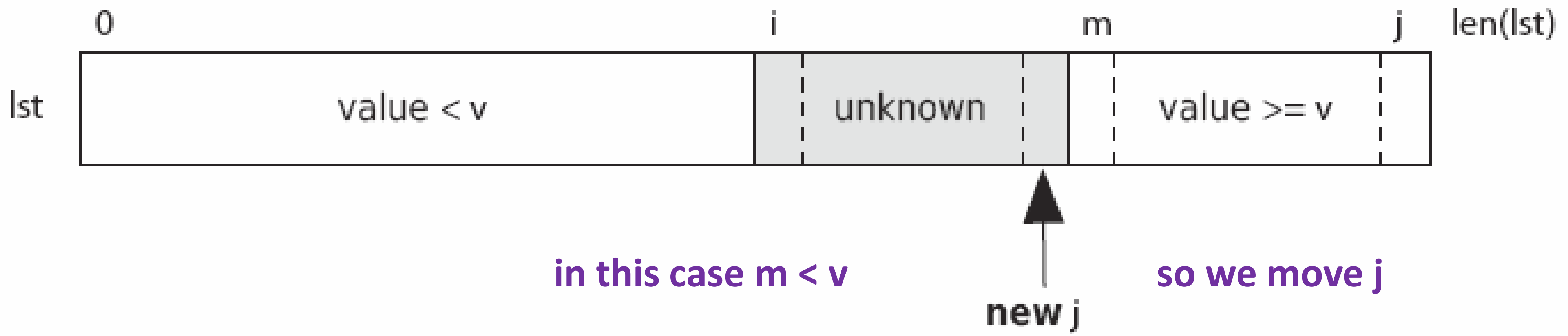

divide list in 2, see if middle value (m) > v or m < v; set i to m if m > v; set j to m if m < v



# Searching

## Searching in a sorted list:

### repeat: divide by two, check value, move either i or j



# Binary Search

```
# binary_search1.py
def binary_search(L, v):
""" (list, object) -> int
Return the index of the first occurrence of value in L, or return
-1 if value is not in L.
>>> binary_search([1, 3, 4, 4, 5, 7, 9, 10], 1) # first one
0
>>> binary_search([1, 3, 4, 4, 5, 7, 9, 10], 4) # twice; get first
2
>>> binary_search([1, 3, 4, 4, 5, 7, 9, 10], 5)  # in middle
4
>>> binary_search([1, 3, 4, 4, 5, 7, 9, 10], 10) # last one
7
>>> binary_search([1, 3, 4, 4, 5, 7, 9, 10], -3) # smaller than all
-1
>>> binary_search([1, 3, 4, 4, 5, 7, 9, 10], 11) # larger than all
-1
>>> binary_search([1, 3, 4, 4, 5, 7, 9, 10], 2)   # not in list, but value between others
-1
>>> binary_search([], -3) # empty list
-1
>>> binary_search([1], 1) # list size = 1
0
"""

# Multiple test cases because many situations can arise, as above
```

# Binary Search

```
 # Mark the left and right indices of the unknown section. (whole list)
    i = 0
    j = len(L) - 1

    while i != j + 1:
        m = (i + j) // 2
        if L[m] < v:
            i = m + 1
        else:
            j = m - 1

    if 0 <= i < len(L) and L[i] == v:
        return i
    else:
        return -1

# doctest with verbose = True
if __name__ == '__main__':
    import doctest
    doctest.testmod(verbose=True)
```

# Binary Search

**Binary search can be much faster than Python's list.index**

| Case | list.index | binary_search | Ratio |
|---|---|---|---|
| First | 0.007 | 0.02 | 0.32 |
| Middle | 105 | 0.02 | 5910 |
| Last | 211 | 0.02 (Wow!) | 11661 |

Table 19—Running Times for Binary Search

**Larger lists do not take much longer for the binary search:**
**- double the list size, needs one more iteration (from 24 to 25 for 20 Million!)**
**- but other methods take twice as long**

**Python has bisect_left and insort_left for inserting into large lists**

# Binary Search Times

## # time_searches2.py

```
# time_searches2.py
import time
import linear_search_1
import linear_search_2
import linear_search_3
import binary_search1
def time_it(search, L, v):
    """ (function, object, list) -> number
    Time how long it takes to run function search to find
    value v in list L.
    """
    t1 = time.perf_counter()
    search(L, v)
    t2 = time.perf_counter()
    return (t2 - t1) * 1000.0

def print_times(v, L):
    """ (object, list) -> NoneType
    Print the number of milliseconds it takes for linear_search(v, L)
    to run for list.index, the while loop linear search, the for loop
    linear search, and sentinel search.
    """
    # Get list.index's running time.
    t1 = time.perf_counter()
    L.index(v)
    t2 = time.perf_counter()
    index_time = (t2 - t1) * 1000.0

    # Get the other four running times.
    while_time = time_it(linear_search_1.linear_search, L, v)
    for_time = time_it(linear_search_2.linear_search, L, v)
    sentinel_time = time_it(linear_search_3.linear_search, L, v)
    Bin_time = time_it(Binary_Search1.binary_search, L, v)

    print("{0}\t{1:>8.1f}\t{2:>8.1f}\t{3:>8.1f}\t{4:>8.1f}\t{5:>8.1f}".format(
        v, while_time, for_time, sentinel_time, Bin_time, index_time))

# modified to make it easier to change values
ListLength = 10000001 # 10 million

L = list(range(ListLength)) # A list of variable length

print  ('Search times in a list of ' , "{:,}".format(ListLength) )
print ('Index\t   while\t     for\t    Sent\t    BinS\t  .index ')
print_times(10, L) # How fast is it to search near the beginning?
print_times(round(ListLength/2), L) # How fast is it to search near the middle?
print_times(ListLength - 50, L) # How fast is it to search near the end?
```

# Binary Search Times

**time_searches2.py output**

```
Search times in a list of  10,000,001
Index      while             for            Sent           BinS      .index
10          0.00            0.00            0.00           0.01        0.00
5000000   897.03          349.20          472.75           0.01       68.98
9999951 1807.62          706.44          942.70           0.01      139.35
```

# Sorting

## When we want to find the largest n or smallest n values in a list?

```
>>> CBA=[563,7590,1708,2142,3323,6197,1985,1316,1824,472,
1346,6029,2670,2094,2464,1009,1475,856,3027,4271,
3126,1115,2691,4253,1838,828,2403,742,1017,613,
3185,2599,2227,896,975,1358,264,1375,2016,452,
3292,538,1471,9313,864,470,2993,521,1144,2212,
2212,2331,2616,2445,1927,808,1963,898,2764,2073,
500,1740,8592,10856,2818,2284,1419,1328,1329,1479]

>>> scopy = sorted(CBA)

>>> scopy[-3:] # return largest 3

>>> scopy[0:3] # return smallest 3
```

## But how does Python sort?

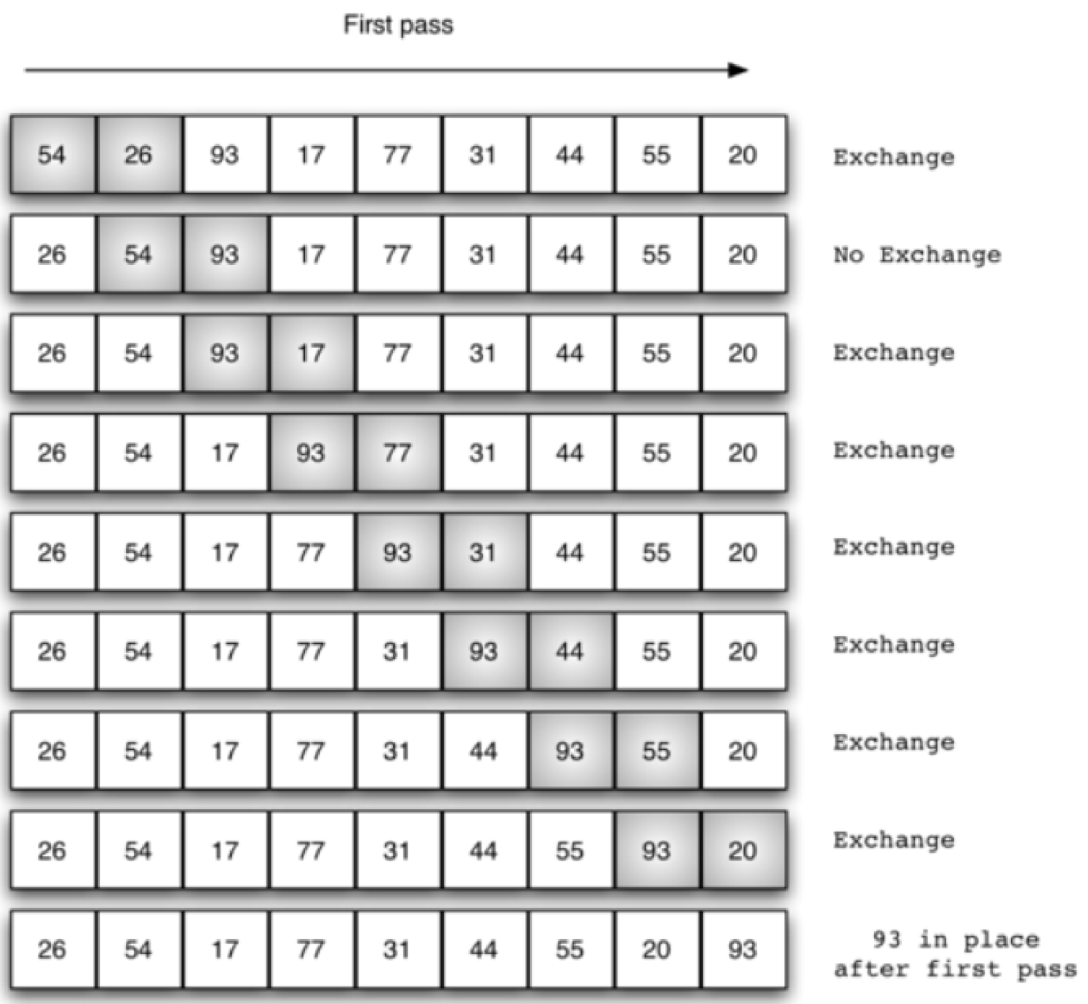
# Bubble Sort

## interactive sorting

https://interactivepython.org/runestone/static/pythonds/SortSearch/toctree.html

5.6 Sorting

5.7 Bubble sort



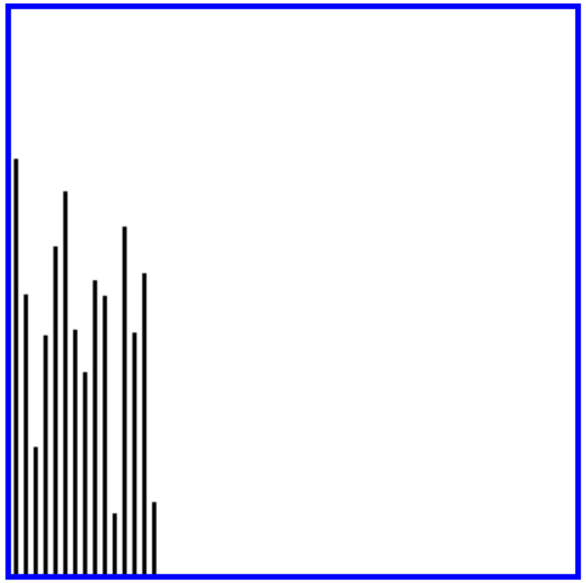
# Bubble Sort

## interactive sorting

https://interactivepython.org/runestone/static/pythonds/SortSearch/TheBubbleSort.html

ActiveCode: 1 The Bubble Sort (lst_bubble)

The following animation shows `bubbleSort` in action.



Initialize | Run | Stop

Beginning | Step Forward | Step Backward | End

To analyze the bubble sort, we should note that regardless of how the items are arranged in the initial list, $n-1$ passes will be made to sort a list of size $n$. Table 1 shows the number of comparisons for each pass. The total number of comparisons is the sum of the first $n-1$ integers. Recall that the sum of the first $n$ integers is $\frac{1}{2}n^2+\frac{1}{2}n$. The sum of the first $n-1$ integers is $\frac{1}{2}n^2+\frac{1}{2}n-n$, which is $\frac{1}{2}n^2-\frac{1}{2}n$. This is still $O(n^2)$ comparisons. In the best case, if the list is already ordered, no exchanges will be made. However, in the worst case, every comparison will cause an exchange. On average, we exchange half of the time.

# Homework

**You don't have to turn anything in.**

https://interactivepython.org/runestone/static/pythonds/SortSearch/toctree.html

**Read the web pages and do interactive things with:**

selection sort

insertion sort

merge sort

quick sort