CS 142

Lecture 3

Analysis, ADTs & Objects

Reminders

Program 1 was assigned

- Due on 1/27 by 11:55pm

Abstraction

When you utilize the `mylist.index(item)` function you are using abstraction

- You don't know how it is implemented, but you do know that it will return an integer (either a -1 to tell you the item is not in the list, or a number from 0 to len(lst)-1 indicating the index of that item.
- "Black-box" – you don't know how it is implemented, but you know how it works

- Always write your code in such a way that you can achieve *implementation independence*
  - You want each function to work independently, so that if you changed the implementation of 1 function, it would not disrupt the rest of your code, or your co-worker's code
- Design programs using a *top-down design*

Measuring Algorithm Efficiency

How do we achieve an analysis of algorithms that is independent of specific implementations, computer used, and data?

Count the number of basic operations of an algorithm, and **generalize** the count.

What are some basic operations?

Problem Size

- Performance is usually measured by the rate at which the running time increases as the problem size gets bigger.
  - Looking at relationship between run-time and problem size
  - Need to first identify problem size
- Example: Analyzing an algorithm that processes a list
  - Problem size = size of list
- In many cases, the problem size will be the value of a particular variable, where the running time of the program depends on how big that value is.

Rhodes College

Counting operations - Exercise

How many operations are required to do the following tasks?

1. Adding an element to the beginning of a list containing n elements.
2. Printing each element in a list containing n elements,
3. Adding a single element to a list using the append() function
4. Performing a nested loop where the outer loop is executed n times, and the inner loop is executed 9 times, e.g.,

```
items = [  [1, 2, 3, 4, 5, 6, 7, 8, 9],
           [1, 2, 3, 4, 5, 6, 7, 8, 9],
           [1, 2, 3, 4, 5, 6, 7, 8, 9],
           [1, 2, 3, 4, 5, 6, 7, 8, 9], … , [ … ]  ]
```

```
sum = 0
for i in items:
    for j in i:
        sum += j

print("sum: ", sum)
```

or

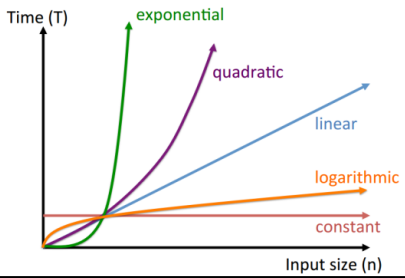
```
sum = 0
for i in range(len(items)):
    for j in range(9):
        sum += items[i][j]

print("sum: ", sum)
```


Algorithm Growth Rate Function

How quickly does time taken by the algorithm increase as a function of the size of the problem?

The growth rate function is a mathematical function used to specify an algorithm's run-time in terms of the size of the problem.



Growth rate function – A or B?

- Consider the following 2 algorithms.

ALGORITHM A

```
n = …
for i in range(n):
    j = 0
    while j < n:
        process(i, j)
        j = j + 5
```

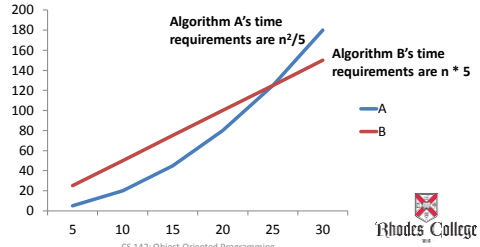
ALGORITHM B

```
n = …
j = 0
for i in range(n):
    while j < 5:
        process(i, j)
        j += 1
```

Rhodes College

Growth rate function – A or B?

n	5	10	15	20	24	25	26	30
A	5	20	45	80	115.2	125	135.2	180
B	25	50	75	100	120	125	130	150



Growth rate function – A or B?

For smaller values of n, the differences between algorithm A ($n^2/5$) and algorithm B(5* n) are not very big. But the differences are very evident for large problem sizes such as when n > 1,000,000

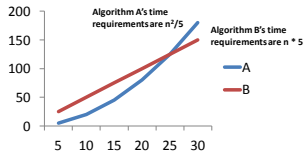
In n is 10^6, then Algorithm A's time is $(10^6)^2 / 5 = (10^{12})/5 = 2 * 10^{11}$
Algorithm B's time is $5 * 10^6$

Bigger problem size produces bigger differences

Algorithm efficiency is a concern for **LARGE** problem sizes.

Big-O Notation

- We use Big-O notation (capital letter O) to specify the complexity of an algorithm, e.g. O(n), $O(n^2)$, $O(n^3)$.
- If a problem of size *n* requires time that is directly proportional to N, the problem is O(n) **(Algorithm B)**
- If time requirement is directly proportional to n^2, the problem is $O(n^2)$ **(Algorithm A)**



Big-O Notation

Only interested in the performance of the program when n is LARGE.

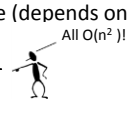
Therefore, we ignore:

- Added constants
- Constant multipliers
- Smaller terms

Constants do not matter in terms of how an algorithm scales.

Big-O Notation

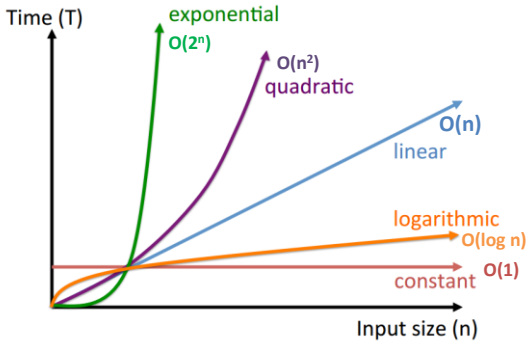
- Compactly describes run-time of an algorithm.
- If your algorithm for sorting a list of *n* numbers takes roughly n^2 operations for the most difficult dataset, then we say that the running time of your algorithm is $O(n^2)$.
- Actual Runtime (depends on your implementation):
  - $-1.5n^2$
  - $-n^2+n+2$
  - $-0.5n^2+1$



Big-O Formal Definition

- When we write that the running time of an algorithm is $O(n^2)$ mean that it does not grow faster than a function with a leading term of $c \cdot n^2$, for some constant c.
- Formally, a function f(n)is Big-O of function g(n), or O(g(n)), when $f(n) \leq c \cdot g(n)$ for some constant c and sufficiently large n.

Comparison of Growth Rates



Examples of Big-O complexities

- **Linear Time - O(n)** – finding the max in a list
- **Logarithmic Time - O(log n)** – Binary search
- **Quadratic Time – $O(n^2)$** – Finding closest pair of points
- **Cubic Time – $O(n^3)$** – Enumerate all triples of elements.
- **Polynomial Time: $O(n^k)$** - Enumerate all subsets of k nodes.
- **Exponential Time – $O(2^{n^k})$** - Enumerate all subsets.

Big-O Notation Practice

What is the Big-O of the following growth functions?

a. $T(n) = n + \log(n)$
b. $T(n) = n^4 + n*\log(n) + 3000n^3$
c. $T(n) = 300n + 60n*\log(n) + 342$

Worst-case and average-case analysis

An algorithm can require different times to solve different problems of the same size. For example, a search for an item in a list.

Best-case analysis: the minimum amount of time that an algorithm requires to solve problems of size n

Worst-case analysis: the maximum amount of time that an algorithm requires to solve problems of size n

Average-case analysis: the average amount of time that an algorithm requires to solve problems of size n

Average performance and worst-case performance are the most commonly used in algorithm analysis.

More Practice

What is the Big-O of the following functions?

```
def exampleA(n):
  s = "PULL FACES"

  for i in range(n):
    print("I must not ", s)

  for j in range(n, 0, -1):
    print("I must not ", s)
```

```
def exampleB(n):
      s = "JUMP ON THE BED"

      for i in range(n):
            for j in range(i):
                  print("I must not ", s)
```

```
def exampleF(n):
  s = "FORGET MY MOTHER'S BIRTHDAY"
  i = n
  while i > 0:
    outF(s)
    i = i // 2

def outF(s):
  for i in range(25, 0, -1):
    print(i, "I must not ", s)
```

```
def exampleD(n):
      s = "PROCRASTINATE"

      for i in range(n):
            for j in range(n, 0, -1):
                  outD(s, n / 2)

def outD(s, b):
      number_of_times = int(b % 10)
      for i in range(number_of_times):
            print(i, "I must not ", s)
```


Object-Oriented Programming (OOP)

- Imagine: You and your programming team have written an extensive customer database program.
  - Original specification said: each customer represented by 3 variables
    - Name
    - Address
    - Phone Number
  - You designed several functions to accept those 3 variables as arguments and perform operations on them.
  - Now your boss decides that a large revision is needed and now all customer information will be stored in a list, instead of 3 variables.

OOP Definitions

- ***Object*** – software entity that contains both data and procedures
  - ***Method*** – functions that perform operations on an object's data attributes.
- ***Encapsulation*** – combining data and code into a single object
- ***Data hiding*** – object's ability to hide its data attributes from code that is outside the object
  - Private vs. public

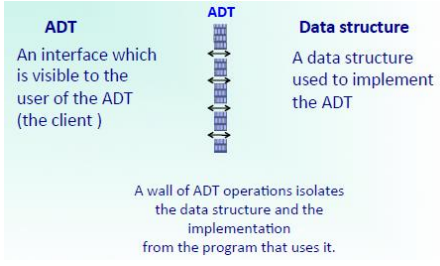
An Everyday Example of an Object

- Data attributes: define the state of an object
  - Example: clock object would have `second`, `minute`, and `hour` data attributes
- Public methods: allow external code to manipulate the object
  - Example: `set_time, set_alarm_time`
- Private methods: used for object's inner workings
  - Example: `increment_hour, increment_minute`

Abstract Data Types (ADTs)

- Programming languages provide:
  - Some concrete data types
    - integers, characters, arrays,…
  - Some abstract data types
    - floating point, lists, tables, two dimensional arrays, records,…
  - Abstract data types are implemented using concrete datatypes
    - (but you don't need to know this to use them)
- Operations are provided for each datatype
  - e.g. creation, assignment, etc.
  - … but you cannot muck around with the internal representations
    - e.g. float is represented in two parts, but you cannot access these directly
  - But: some languages do allow you access to the internal representations
    - e.g. in C, you can use pointers to access the internals of arrays
    - this removes the distinction between the abstraction and the implementation
    - it destroys most of the benefits of abstraction
    - it causes confusion and error

An ADT is Not a Data Structure



Disadvantages/Advantages of an ADT

Disadvantages

initially there is more to consider, design issues, code to write and maintain, overhead of calling a method to access ADT information, greater initial time investment.

Advantages

A **client** (the application using the ADT) doesn't need to know about the implementation,

Maintenance of the application is easier,

The programmer can focus on problem solving and not worry about the implementation.

Designing an ADT

- Questions to ask
  - What data does a problem require?
  - What operations does a problem require?

No need to ask HOW we are storing the data

No need to ask HOW we are implementing the operations

List ADT

A list needs to hold an unknown number of items
Attributes:
values in list
Operations:
- add an item to end of list
- insert item into middle of list
- remove item from list
- sort a list
- find item in list
- print items in list

Designing a BankAccount ADT

- What sort of things should we be able to do?

ADTs and Objects

- ***Abstract data type (ADT)*** – collection of functions or methods that manipulate an underlying representation
  - Think of this as pseudo-code
- ***Class***: code that specifies the data attributes and methods of a particular type of object
- ***Instance:*** an object created from a class
  - There can be many instances of one class


```
class BankAccount(object):

    #The __init__ method accepts an argument for
    #The account's initial balance. It is assigned
    #to the __balance attribute.
    def __init__(self, bal):
        self.__balance = bal

    #Makes a deposit into the account
    def deposit(self, amount):
        self.__balance += amount

    #Withdraws an amount from the account.
    def withdraw(self, amount):
        if self.__balance >= amount:
            self.__balance -= amount
        else:
            print('Error: Insufficient funds')

    #Returns the account balance
    def get_balance(self):
        return self.__balance
```

←Class Definition (using the new-style)
Inherits from the built-in class object.

`self` parameter: required in every method in the class – references the specific object that the method is working on


```
class BankAccount(object):

    #The __init__ method accepts an argument for
    #The account's initial balance. It is assigned
    #to the __balance attribute.
    def __init__(self, bal):
        self.__balance = bal

    #Makes a deposit into the account
    def deposit(self, amount):
        self.__balance += amount

    #Withdraws an amount from the account.
    def withdraw(self, amount):
        if self.__balance >= amount:
            self.__balance -= amount
        else:
            print('Error: Insufficient funds')

    #Returns the account balance
    def get_balance(self):
        return self.__balance
```

Initializer method: automatically executed when an instance of the class is created


```
class BankAccount(object):

    #The __init__ method accepts an argument for
    #The account's initial balance. It is assigned
    #to the __balance attribute.
    def __init__(self, bal):
        self.__balance = bal

    #Makes a deposit into the account
    def deposit(self, amount):
        self.__balance += amount

    #Withdraws an amount from the account.
    def withdraw(self, amount):
        if self.__balance >= amount:
            self.__balance -= amount
        else:
            print('Error: Insufficient funds')

    #Returns the account balance
    def get_balance(self):
        return self.__balance
```

An object's data attributes should be private

2 underscores before variable name designates **private** variable in Python

```
class BankAccount(object):

    #The __init__ method accepts an argument for
    #The account's initial balance. It is assigned
    #to the __balance attribute.
    def __init__(self, bal):
        self.__balance = bal

    #Makes a deposit into the account
    def deposit(self, amount):
        self.__balance += amount

    #Withdraws an amount from the account.
    def withdraw(self, amount):
        if self.__balance >= amount:
            self.__balance -= amount
        else:
            print('Error: Insufficient funds')

    #Returns the account balance
    def get_balance(self):
        return self.__balance
```

Accessor method: return a value from a class's attribute without changing it

- –Safe way for code outside the class to retrieve the value of attributes


```
class BankAccount(object):

    #The __init__ method accepts an argument for
    #The account's initial balance. It is assigned
    #to the __balance attribute.
    def __init__(self, bal):
        self.__balance = bal

    #Makes a deposit into the account
    def deposit(self, amount):
        self.__balance += amount

    #Withdraws an amount from the account.
    def withdraw(self, amount):
        if self.__balance >= amount:
            self.__balance -= amount
        else:
            print('Error: Insufficient funds')

    #Returns the account balance
    def get_balance(self):
        return self.__balance
```

Mutator methods: store or change the value of a data attribute