Part 1

INTRODUCTION TO DATA STRUCTURES IN PYTHON

Course Web Site

https://samyzaf.com/braude/DSAL

Introduction to:

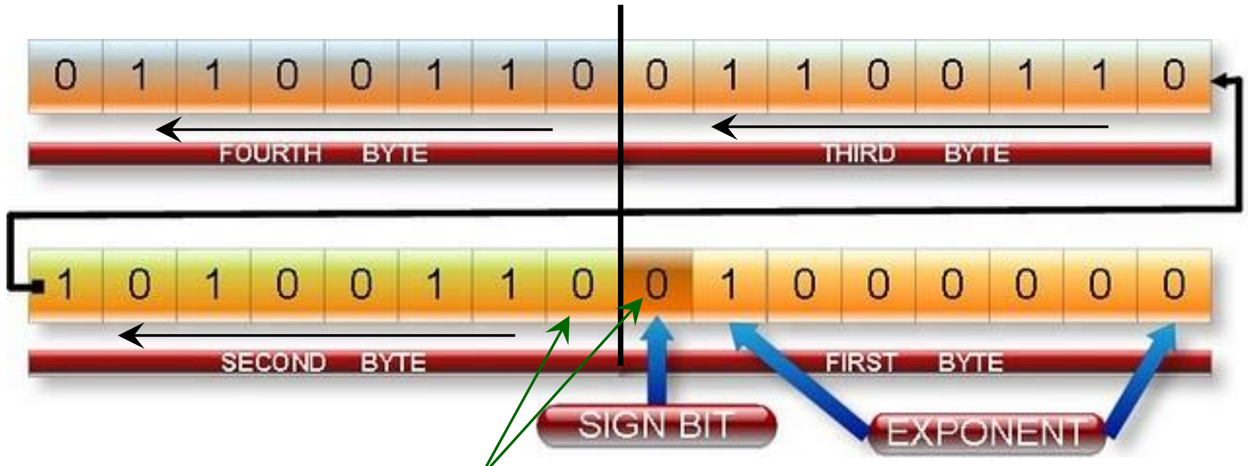
DATA STRUCTURES AND ALGORITHMS

Data Structures

- Systematic methods for organizing information in a computer
- A **data type** consists of the values it represents and the operations defined upon it
- In the **C** programming language, a data type is usually represented by the **struct** concept.
- But the **struct** represents only the data type values and does not describe what kind of operations can be applied on the data type
- In object oriented languages, the **class** concept extends the **struct** concept by also adding methods that can be applied on a data type

Data Type Binary Representation

- Data types may be viewed in several ways:
  - As **abstract entities**
  - As **concrete implementations**
- For example, there are many ways to represent a floating number like x=5.2 – here is one common way to do it (32 bit arch):



exp = e +127
See comment
Below slide

5.2 = 5 + 0.2 = 101 + 0.0011001100110011001100110011…
= 101.0011001100110011001100110011…
= + 1.0100110011001100110011001100110011… * 2^2

Data Type Binary Representation

- Note that some data types may not have a fully accurate representation!
- For example, the float number x=5.2 is not really equal to its binary representation above! Moreover, it will have a different value in a 64 bit architecture!
- This is however will not concern us in this course as we're more concerned with the **abstract view** of data types!
- Binary representations of data types is the business of other courses and not ours!
- We do however need to be aware of the basic ideas of representations in order to be able to do realistic analysis of algorithms, estimate input and output sizes, estimate space and run time figures

Abstract Data Type (ADT)

- An **abstract data type** (**ADT**) is a programmer-defined data type that specifies a set of data values and a collection of **well-defined operations** that can be performed on those values
- Only the formal definition of the data type is important and **NOT** how it is implemented in binary form or in hardware
- This is sometimes called:
  "Separation of Interface and Implementation"
- **Information Hiding** – how the data is represented and how the operations are implemented is completely <u>irrelevant</u> when we define a new **A**bstract **D**ata **T**ype (**ADT**) !

Example: String ADT

```
String Data Type:
    An string of characters like
    s = "Hello World"
    s = "Guido Van Rossum, 1993"

Operations:
    upper(s)            All characters to upper case
    lower(s)            All characters to lower case
    find(s,w)           Find a word w in s (return index)
    replace(s,w1,w2)    Replace sub word w1 with w2
```

EXAMPLE CODE:

```
s = "Hello World"
upper(s) = "HELLO WORLD"
lower(s) = "hello world"
find(s, "Wo") = 6
replace(s, "lo", " NEW") = "Hel NEW World"
```

ADT As Interface Design

- Note that the term ***"string of characters"*** does not imply anything about its implementation (how English characters are represented?)
- It can be implemented as a C **array of characters** terminated by a **NULL**
- It can be implemented like a Java or C++ String object
- We may even decide to encode and compress the string if it size is too large
- We can decide to break each string to chunks of 4K in different memory locations and keep a central table for accessing these chunks, etc …

ADT As Interface Design

- Similarly, nothing on how the **find()** and **replace()** algorithms should be implemented is mentioned!
- All we care is about how we **<u>Interface</u>** with the string data type? (How to do? instead of how it is done?)
- All implementation issues are **<u>irrelevant</u>** to the ADT specification!

Algorithms

- After defining an ADT we will proceed to the second part of our course: ***ALGORITHMS***
- Named after the mathematician Muḥammad ibn Mūsā al-Khwārizmī (Bagdad 780-850) which invented the concept and the first mathematical algorithms (including an algorithm for solving quadratic equations)
- **ALGORITHM:**
  - An effective method expressed as a finite list of well-defined instructions for calculating a function (Wikipedia)
  - Simply put, a ***data structure*** is a systematic way of organizing and accessing data, and an ***algorithm*** is a step-by-step procedure for performing some ***task*** in a finite amount of time (Goodrich/Tamassia/Goldwasser book)

أبو عبد الله محمد بن موسى الخوارزمي

Example: Euclid’s GCD Algorithm

- **GCD = Greatest Common Divisor**
- Perhaps one of the most famous algorithms in history
- Formulated by Euclid around 300 BC (without knowing the algorithm concept)
- **Problem:** given two integers A and B, find the largest integer G which divides both A and B
- Here is the most naïve way to solve the problem:

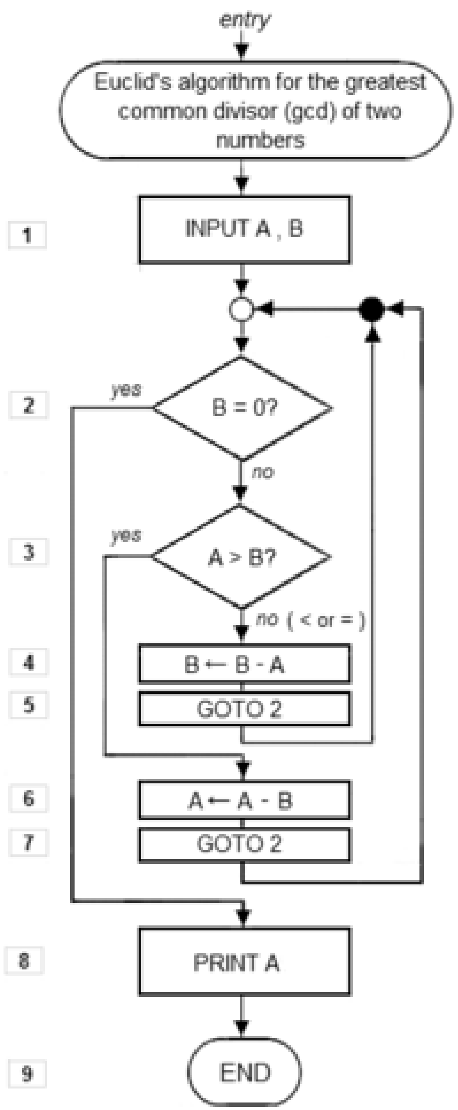
```
def gcd1(a, b):
    if a == 0: return b
    if b == 0: return a
    m = min(a,b)
    greatest = 1
    d = 1
    while d <= m:
        if a%d == 0 and b%d == 0:
            greatest = d
        d += 1
    return greatest
```

Flow Charts

- Modern algorithms are often written as “Flow Charts” as the figure on the right side which describes Euclid’s algorithm

- There are many graphical computer programs for drawing beautiful Flow Charts which you can use for designing your algorithms

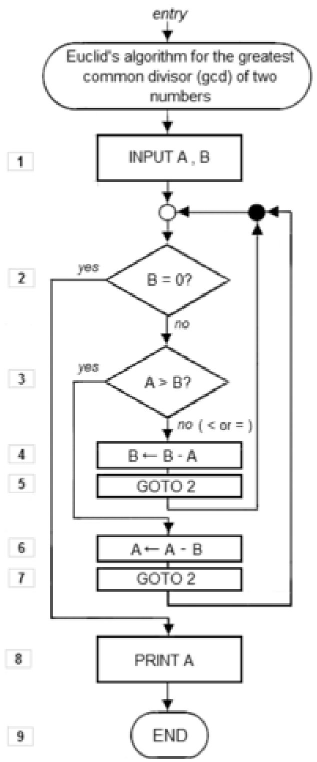
- Here is a Flow Chart for a popular version of Euclid’s Algorithm:



Euclid’s GCD Algorithm in Python

- The other method for expressing Algorithm is by a semi-formal language called Pseudo-Code
- Since Python is simple and very readable as pseudo-code and at the same time it is also a fully running formal language, there are more and more courses and books that use it for a data structures and algorithms courses

```
def gcd2(a, b):
    if b == 0:
        return a
    else:
        if a>b:
            a = a-b
        else:
            b = b-a
        return gcd2(a,b)
```



Euclid's GCD Algorithm: Correctness Proof

- Theorem: Assume that a>b>0, are two integers.
  For any integer **d**: **d** divides **a** and **b** ⇔ d divides a-b and b
- Proof is easy!
- Definition: div(a,b) = {d | d divides a and b}
- Consequence: div(a,b) = div(a-b, b)
- Consequence: gcd(a,b) = gcd(a-b, b)

Euclid's GCD Algorithm in Python Recursive Algorithm

However the gcd2 is recursive, and thus can fail if a and b are very large:

```
def gcd2(a, b):
    if b == 0:
        return a
    else:
        if a>b:
            a = a-b
        else:
            b = b-a
        return gcd2(a,b)
```

Problem with recursion:

```
RuntimeError: maximum recursion depth exceeded in cmp
```

Euclid's GCD Algorithm in Python
Non-recursive Algorithm

```python
def gcd3(a, b):
    "Find the greatest common divisor for two integers: a,b"
    if a == 0:
        return b
    elif b == 0:
        return a
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

Python's GCD Algorithm

- Python contains an official **GCD** algorithm as part of the **fractions** module:

```
def gcd(a,b):
    while a:
        a, b = b%a, a
    return b
```

- This follows immediately from: gcd(a,b) = gcd(a, b-a)
- For any integer k, gcd(a,b) = gcd(a, b – ka) = gcd(b-ka, a)
- If k = b/a, then b-ka = b%a, and we get: gcd(a,b) = gcd(b%a, a)
- Why the algorithm must stop? (could be an infinite loop?)
  Prove that the numbers are decreasing until a==0

Run Time Analysis

```
import time

def gcd_time_test(f, a, b):
    print "Running %s(%d,%d)" % (f.func_name, a,b)
    start = time.time()
    try:
        print "gcd =", f(a,b)
    except Exception as e:
        print e
    end = time.time()
    print "runtime = %.3f seconds" % (end-start,)
```

Which Algorithm is the fastest?

This is just a simple performance test. A more rigorous test should sample a larger variety of numbers and each calculation should be repeated several times (average time)

```
def test1():
    a = 2**13 * 3**4 * 5**3
    b = 2**7 * 3**5 * 5**2
    gcd_time_test(gcd1, a, b)
    gcd_time_test(gcd2, a, b)
    gcd_time_test(gcd3, a, b)
    gcd_time_test(gcd4, a, b)
```

Example 2: Primality

- Data Type: unsigned integers: **0, 1, 2, 3, 4, 5, …**
- Definition: a **prime number** is an integer p>1 which has exactly two divisors: 1, and p.
- Problem: Given a positive integer **n**, find if **n** is a prime number?
- Here is a Naïve simple algorithm that solves this problem:

```
def is_prime(n):
    if n <=1: return False
    i=2
    while i<n:
        if n%i==0:
            return False
        i += 1
    return True
```

Container/Collection Terminology

- In Object Oriented Design, a **Container** is any object that contains other objects in itself
- Other words: a collection is a group of values with no implied organization or relationship between the individual values (Rance Necaise book)
- Some languages restrict the elements to a specific data type such as integers or floating-point values
  - Python collections do not have such restriction

Collection Types

- The programming languages and literature are full with many such object with many different names
  - **List**
  - **Array**
  - **Sequence**
  - **Vector**
  - **Set**
  - **Stack**
  - **Queue**
  - **Heap**
  - **Map**
  - **Hash Table**
  - **Dictionary**
  - **Tree**
  - **Graph**
  - **Multimap**
  - **Multiset**
  - **Priority Queue**
  - **String**

Leaf Objects

- In contrast to Container object, a Leaf Object is an object that does not contain any reference to other objects ("has no child objects")
- In Python these are sometimes called "primitive types"
  - Integer
  - Float
  - Complex number
  - Boolean
- Leaf Objects are the building blocks from which all other objects are built

Primitive Types

- Integer: -5, 19, 0, 1000 (C long)
- Float: -5.0, 19.25, 0.0, 1000.0 (C double)
- Complex numbers: a+bj
- Boolean: True, False
- Long integers (unlimited precision)
- Immutable string: “xyz”, “Hello, World”

Arithmetic Operations

Operation	Result
x + y	sum of *x* and *y*
x - y	difference of *x* and *y*
x * y	product of *x* and *y*
x / y	quotient of *x* and *y* (Integer division if x,y integers
x % y	remainder of x / y
-x	*x* negated
+x	*x* unchanged
abs(x)	absolute value or magnitude of *x*
int(x)	*x* converted to integer
long(x)	*x* converted to long integer (this is very long …)
float(x)	*x* converted to floating point
complex(re,im)	a complex number with real part *re*, imaginary part *im*. *im* defaults to zero
c.conjugate()	conjugate of the complex number *c*. (Identity on real numbers)
divmod(x, y)	the pair (x / y, x % y)
pow(x, y)	*x* to the power *y*
x ** y	*x* to the power *y*

Comparisons

Operation	Meaning
<	strictly less than
<=	less than or equal
>	strictly greater than
=>	greater than or equal
==	equal
!=	not equal
is	object identity
is not	negated object identity

Bitwise Operations

<table>
<tr><th>Operation</th><th>Result</th></tr>
<tr><td>x | y</td><td>bitwise or of x and y</td></tr>
<tr><td>x ^ y</td><td>bitwise exclusive or of x and y</td></tr>
<tr><td>x & y</td><td>bitwise and of x and y</td></tr>
<tr><td>x << n</td><td>x shifted left by n bits</td></tr>
<tr><td>x >> n</td><td>x shifted right by n bits</td></tr>
<tr><td>~x</td><td>the bits of x inverted</td></tr>
</table>

The Complex Numbers Class

- The **cmath** module defines Complex numbers arithmetic
- Python contains a built-in type (class) for complex numbers
- A complex number object has two fields and one method:

imag	imaginary part
real	real part
conjugate()	The conjugate number

```
import cmath
z = cmath.sqrt(-9)
⇒ 3j
z = cmath.sqrt(5-12j)
⇒ (3-2j)
z.imag
⇒ -2.0
z.real
⇒ 3.0
z.conjugate()
⇒ (3+2j)
```

Abstract Data Types Operations

Constructors	Methods for creating new objects
Accessors	Methods for accessing internal data fields without modifying the data!
Mutators	Methods for modifying object data fields
Iterators	Methods for processing data elements sequentially

List Abstarct Data Type
Procedural Design – part 1

- `L = list_create1(e0, e1, e2,... ,en-1)`
  - Create a new list **L** from n elements: **e0, e1, …, en**
- `L = list_create2(other)`
  - Create a new list **L** from other list or another container structure
- `get_item(L,i)` - Get element **i** of list **L**
- `set_item(L,i,e)` - Set element **i** of list **L** to **e**
- `contains(L,e)`
  - Check if element **e** belongs to list **L.** Returns: Boolean **True** or **False**
- `append(L,e)`
  - Add a new element **e** to **L**
  - What if **e** already belongs to **L**? (answer: duplications are allowed!)
- `remove(L,e)`
  - Remove an element **e** from **L**
  - What if **e** is not in **L**? (two possibilities: 1. do nothing, 2. raise an error)

The List ADT
Procedural Design – part 2

- `insert(L, index, e)`
  - Insert a new element **`e`** at index **`index`**
  - Side effect: list grows by one element
- `size(L)`
  - Return the size of **`L`**
- `extend(L,L2)`
  - Extend list **`L`** by list **`L2`**
- `reverse(L)`
- `slice(L,i,j)`
  - Return a sub-list consisting of all elements of **`L`** from index **`i`** to index **`j-1`**
- `index(L,e)`
  - Find the index of element **`e`** in **`L`**

Test Driven Development

- In this highly recommended methodology you write your tests before the implementation of your ADT !!!
- After implementation, your tests should run and **PASS** after each modification you make to your implementation ("nightly test regression")
- The following tests are your "insurance policy" that your implementation is correct. The more tests you write, the better you're insured

```
# Testing our List ADT
L1 = list_create1(2, 3, 5, 7, 11)
L2 = list_create2(L1) # copy constructor
assert L2 == L1       # Assertion
append(L1, 37)
remove(L1, 2)
remove(L1, 3)
L3 = list_create1(5, 7, 11, 37)
assert L1 == L3       # Assertion
```

ADT Implementation

- After defining an abstract data type, we need to implement it in a specific programming language
- First we must define a concrete data structure in the particular language for representing our abstract data
- Python basic data structures are usually implemented in the C programming language
- More complex data structures are usually implemented over the Python languages itself, and later transformed to C code if performance is critical

Python List ADT Implementation

```
typedef struct {
    int ob_refcnt ;
    struct _typeobject *ob_type ;
    int ob_size ;
    PyObject **ob_item ;
    int allocated ;
} PyListObject ;
```

- Lists in Python are implemented as a **C <u>array</u> of PyObject pointers**
- `**ob_item` is an **<u>array</u>** of pointers to **PyObject pointers**
- A Python list is therefore an array of references to any **Python objects**!
- A `PyListObject` can grow and shrink (so there could be many calls to `malloc` and `free` on the way … but Python users shouldn't care)

C Implementation of append

```c
static int app1(PyListObject *self, PyObject *v) {
    Py_ssize_t n = PyList_GET_SIZE(self) ;

    assert (v != NULL) ;
    if (n == PY_SSIZE_T_MAX) {
        PyErr_SetString(PyExc_OverflowError,
            "cannot add more objects to list") ;
        return -1 ;
    }

    if (list_resize(self, n+1) == -1)   /* increase list size by +1 */
        return -1 ;

    Py_INCREF(v) ;                      /* incr reference count of v */
    PyList_SET_ITEM(self, n, v) ;       /* add pointer v at the end */
    return 0 ;
}
```

C Implementation of insert

```c
static int ins1(PyListObject *self, Py_ssize_t where, PyObject *v) {
    Py_ssize_t i, n = Py_SIZE(self) ;
    PyObject **items ;
    if (v == NULL) {
        PyErr_BadInternalCall() ; return -1 ;
    }
    if (n == PY_SSIZE_T_MAX) {
        PyErr_SetString(PyExc_OverflowError,  "cannot add more objects to list") ;
        return -1 ;
    }
    if (list_resize(self, n+1) == -1)
        return -1 ;
    if (where < 0) {
        where += n ;
        if (where < 0)
            where = 0 ;
    }
    if (where > n)
        where = n ;
    items = self->ob_item ;
    for (i = n ; --i >= where ; )          /* Move all items [i:n] to [i+1:n+1] ! */
        items[i+1] = items[i] ;
    Py_INCREF(v) ;
    items[where] = v ;                     /* insert the new value v at index where */
    return 0 ;
}
```

No time in class
Home reading!

C Implementation of list reverse

```
/* Reverse a slice of a list in place, from lo to hi (exclusive) */
static void reverse_slice(PyObject **lo, PyObject **hi) {
    assert(lo && hi) ;    /* make sure lo and hi are not NULL */
    PyObject* tmp
    --hi ;                /* hi itself is excluded */
    while (lo < hi) {
        tmp = *lo ;
        *lo = *hi ;
        *hi = t ;
        ++lo ;
        --hi ;
    }
}
```

List Reverse Implementation (2)
procedural design, Python, Recursive

```
def _reverse_recursive(S, begin, end):
    """ Reverse elements in slice S[begin:end+1] """
    if end>begin:
        # swap first and last elements
        S[begin], S[end] = S[end], S[begin]
        # Recursion:
        _reverse_recursive(S, begin+1, end-1)

def reverse_recursive(S):
    _reverse_recursive(S, 0, len(S)-1)
```

Reverse Implementation (3)

procedural design, Python, Iterative

```python
def reverse_iterative(S):
    """ Reverse elements in sequence S."""
    a, b = 0, len(S)-1
    while a < b:
        S[a], S[b] = S[b], S[a]
        a, b = a+1, b-1
```

```
Example:
S = [0, 1, 2, 3]
a, b = 0, 3  ==>  [3, 1, 2, 0]
a, b = 1, 2  ==>  [3, 2, 1, 0]
a, b = 2, 1  ==>  done
```

Testing your implementation

- Remember: tests must be written before you even think about an implementation!
- Make sure your tests cover the major features
- After writing an implementation you must run your tests: if they fail, then your implementation is bad
- After changing an implementation you must run all the tests again
- You may decide to throw away the whole implementation and write a new one, without any change to your ADT specification (“same Interface different implementation”) – your tests should pass again with the new implementation!

Interface and Implementation
Totally Separated Things !!!

- There should be a **total separation** between an **ADT** specification (sometimes called “Interface specification”) and its possibly many implementations
- For example, the Python Language has a full implementation over Java (called **Jython**), and at the same time Microsoft has a full implementation of Python over C# which is called **IronPython**
- The Python implementation over **C** is called **CPython**
- The same **Python tests** must all pass in all three implementations: **CPython**, **Jython**, and **IronPython** !
- The Python language itself is a pure interface! Unlike low level languages such as C it does not have any business with hardware registers, contiguous memory cells, etc. No relation to hardware at all!

Problems with Procedural Design

- No clear separation between major and minor data types
- For example, when we see `append(a,b)` it's not always clear which is the list and who is the element?
- Composite expressions like:

```
insert(append(extend(L,L2),a3),7,b4)
```

  can be very hard to read and understand
- Generic method names like `append()`, `insert()`, `remove()`, `size()`, etc., cannot be reused for a different data structure (like `FILE` or `Vector`), since they are global and already taken by the List data type … this is a serious trouble.
- Code reuse is difficult

The List ADT
Object Oriented Design – part 1

- `L = list_create1(e0, e1, e2,..., en-1)` **[constructor]**
  - Create a new list `L` from n elements: `e0, e1,... , en-1`
- `L = list_create2(other)` **[constructor]**
  - Create a new list **L** from other list or a container structure
- `L.item(i)` - Get element `i` of list `L` **[accessor]**
- `L.contains(e)` **[accessor]**
  - Check if element **e** belongs to list `L`
  - Returns: boolean `True` or `False`
- `L.append(e)` **[mutator]**
  - Add a new element `e` to `L`
  - What if **e** already belongs to `L`? (answer: duplications are allowed!)
- `L.remove(e)` **[mutator]**
  - Remove an element `e` from `L`
  - What if `e` is not in `L`? (two possibilities: 1. do nothing, 2. raise an error)

The List ADT
Object Oriented Design – part 2

- `L.replace(index, e)` [mutator]
  - Replace element at index **`index`** with **`e`**
- `L.insert(index, e)` [mutator]
  - Insert a new element **`e`** at index **`index`**
  - Side effect: list grows by one element
- `L.size()` [accessor]
  - Return the size of **`L`**
- `L.extend(L2)` [mutator]
  - Extend list **`L`** by list **`L2`**
- `L.reverse()` [mutator]
- `L.slice(i,j)` [accessor]
  - Return a sub-list consisting of all elements of **`L`** from index **`i`** to index **`j-1`**
- `L.index(e)` [accessor]
  - Find the index of element **`e`** in **`L`**

Test Driven Development

- We need to update all our procedural oriented test to be object oriented

```
# Testing our List ADT
L1 = list_create1(2,3,5,7,11)
L2 = list_create2(L1)      # “copy constructor”
assert L2 == L1            # Assertion
assert L2.item(0) == 2
L1.append(37)
L1.remove(2)
L1.remove(3)
L3 = list_create1(5,7,11,37)
assert L1 == L3            # Assertion
assert L3.index(37) == 3   # Assertion
L3.reverse()
L4 = list_create1(37,11,7,5)
assert L3 == L4            # Assertion
```

Procedural notation

- The functional notation
  `foo(x)`, `bar(x,y)`, `baz(x,y,z)`
  was invented by the Mathematician Leonard Euler at 1748
- There is no specific sacred or holly reason for this notation! Euler could at the same time use ‘<x>f’ or ‘f-x-’ or many other possible notations
- We already have exceptions to this rule when we write x+y instead of add(x,y), or x**n instead of power(x,n).
- Python writes: `L = [a, b, c]` instead of `list_create(a,b,c)`

Python List Constructors

- The most basic constructor for lists is:

```
L = [x0, x1, x2, ..., xn]
```

  - It corresponds to: `list_create1(x0, x1, x2, …, xn)`
- The other constructor is list(container_object)
- Lists can be created from a variety of other container objects such as: set, array, dictionaries, and other list

Naming Issues

- Specification name and Implementation name do not have to be the same!
- For example, in Python, the call

```
L = list_create1(e0, e1, e2,..., en-1)
```

  has been changed to:

```
L = [e0, e1, e2, …, en-1]
```

  and the call

```
L.contains(e)
```

  Has been changed to:

```
e in L
```

- The only essential thing is that the name conveys the meaning of the operation, and the operation is precisely defined

Python List Syntactic Sugar

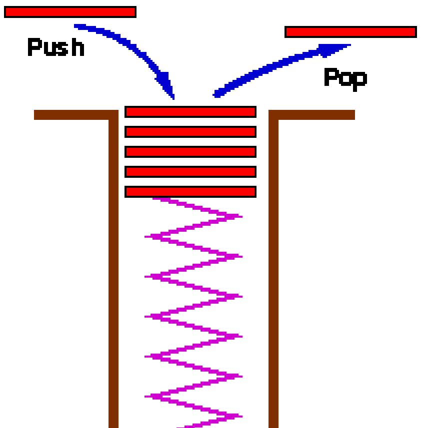
Operation	Python Syntactic Sugar
`L=list_create1(a,…,b)`	`L = [a, ...,b]`
`L=list_create2(other)`	`L = list(other)`
`L.contains(e)`	`e in L`
`L.item(i)`	`L[i]`
`L.size()`	`len(L)`
`L.slice(i,j)`	`L[i:j]`
`L.equals(other)`	`L == other`
`L.remove_by_index(i)`	`del L[i]`
`L1.add(L2)`	`L1+L2`
`L.mul(n)`	`L*n  or  n*L`

A Word About Destructors

- Some object oriented languages (like C++) contain an additional method type: **destructor**
- A **destructor** is a method for destroying (or terminating) an object
- A destructor usually frees the memory that was used by the object and may also perform additional cleanup and finalization tasks
- In such languages, failure to delete objects at the right time can lead to serious memory problems, and even to program crash
- Modern object oriented languages such as **Java**, **C#**, and **Python**, contain a mechanism (called “garbage collection”) which automatically deletes objects as soon as they’re not needed anymore
- We will therefore not bother about this concept anymore in this course
- In extreme cases if needed you can use the Python del operator to delete objects: `del L`

Stack Abstract Data Type
Description

- Sequence type (container) in which elements are pushed and popped out from the top end
- AKA LIFO – Last In First Out



Stack Abstract Data Type
Interface

- `s = Stack()` **Constructor**
  - Create a new empty stack
- `s.push(item)` **Mutator**
  - Add an item to the top of the stack
- `s.pop()` **Mutator**
  - Pop an item to the top of the stack
- `s.peek()` **Accessor**
  - Return the item to the top of the stack (don't pop it!)
  - Return `None` if stack is empty (this is not a good idea, why?)
- `s.size()` **Accessor**
  - Return the number of items in the stack
- `s.is_empty()` **Accessor**
  - Return `True` if stack is empty, `False` if stack is non-empty

Stack Test 1

```
s = Stack()
s.push(1)
s.push(1)
s.push(2)
assert s.pop() == 2
assert s.pop() == 1
assert s.pop() == 1
assert s.is_empty()
```

Stack Test 2

```
s = Stack()
expression = "a+(b*(c+d)+x*(y-a)+z)-n“

# Check if left/right parens are
# legally balanced
for char in expression:
    if char == '(':
        s.push('L')
    if char == ')':
        if s.peek() == 'L':
            s.pop()
        else:
            s.push('R')

assert s.is_empty()
```

Stack Test 2: Stack Frames

```
s = Stack()
expression = "a+(b*(c+d)+x*(y-a)+z)-n“

Frame 0: empty stack
Frame 1: L
Frame 2: L, L
Frame 3: L
Frame 4: L, L
Frame 5: L
Frame 6: empty stack
```

Stack Implementation

```
class Stack :
  def __init__(self) :
    self.items = []

  def push(self, item) :
    self.items.append(item)

  def pop(self) :
    return self.items.pop()

  def peek(self):
    return self.items[-1]

  def is_empty(self) :
    return (self.items == [])
```

http://www.greenteapress.com/thinkpython/thinkCSpy/html/chap18.html