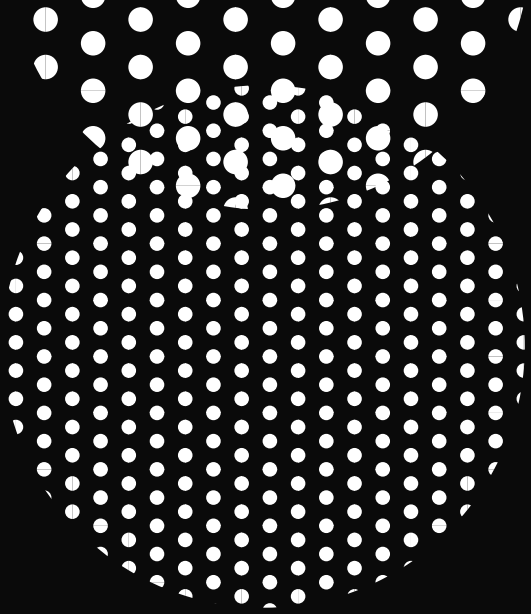


Short Circuit error

- If (3/0 == 1 or 3/2 != 0)

  undefined, will create an error

- If (3/2 != 0 or 3/0 == 1)

  This is true. Therefore, the back portion will be ignored and the result is TRUE

Chained Variables

- When a = b = c = 10, assign all variables to the last indicator

Swapping Variables

- You can use

```
a = temp
a = b
b = temp
```

a, b = b, a

Formatting Numbers

- format (var, '___')

0.2f = To 2 d.p. → 1.35
% = Put in % → 135.236%
.0% = Round off and % → 135%

d = designator
,d = 123,456
10d = ---------- 12345

You can also use {}

```
print("lol {:} I am {0}".format("hi", "poop"))
```

lol poop I am hi

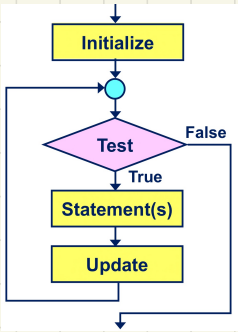
```
print("{0} ate {1:.1f} of cake and {nom}".format("tom", 3/100, nom="lol"))
```

tom ate 66.7% of cake and lol

Looping

- Can dynamically change how many times to repeat

4 Basic steps



1. Initialize: Loop control varible, how many times to loop?
2. Test: Do we continue the loop or not?
3. Loop Body: If evaluated = True, execute internal
4. Update: Modify loop control varible so next test can exit loop [num = num + 1]

How to loop?

1. ID the statement that needs to loop
2. Wrap statement in loop
3. Code the loop continuation and the addition function

Counter Control	Sentinel Control
- Loop repeats N times * fixed loop number	Total loop number is related to the data provided. Exact number of loops unknown. ↳ you can use Sentinel varibles like time to cut off a counter

Sentinel Value

→ Make sure they don't appear in normal inputs

- This is the "termination condition"

While

- Allows repetition as long as a condition is true

  While <boolean expression>
      Suite(intented statements)

- When condition becomes false, the loop exits.

Else for while loop

When while loop = False, else suite kicks in

- Will start even if while was never true
- Final task when loop ends

Continue

- Skip the rest of the while & move back to start

Break for while loop

* non-normal exit / early exit

- Immediately escape the while loop, skip pass the else suite as well

```
value = 1
print ("before while", value)

while value <=3:
    value = value + 1
    print ("while", value)
    if value == 2:
        break;
else:
    print ("else", value)

print ("after while", value)
```

Compared to quick check question: added **if** and **break**.

- Executing break exits the immediate loop that contains it.
- It goes after the whole enclosing loop

For Statement

- Performs assignment of (in) to (var).

aka, `for the_char in "imfao"`
`print(the_char)`

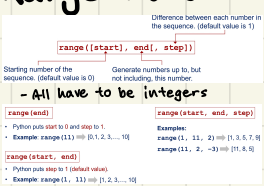
Will output: i m f a o

* usage of [ ]
- Denotes grouping.
`["xyz"]` = xyz
`"xyz"` = x y z

- Basically, assigns information to the Var until all elements have been assigned

Continue, Break and Else will also function in FOR loops.

Range function



- All have to be integers

Nested loops

- One loop inside another

Pass Statement

- Test to see if it is correct
- Placeholder, come back later to fill details

"String"

★ To put apostrophe, you can use escape character

'Mike\'s book'

- In a sequence, can identify with unique index (position)

Index from

Left → ① Non-ve value, Start @ 0

Right → ② -ve value, Start @ -1

String = "Hello World"

Characters	H	e	l	l	o		W	o	r	l	d
Indices	0	1	2	3	4	5	6	7	8	9	10
										…	−2	−1

*Indices 10 and −1 point to the same location: **d***

```
myStr = "Hello World"
x = myStr[1]
print(x)          → #will print e
print(myStr)      → ? #will print Hello World
print(myStr[-2])  → ? #will print l
print(myStr[11])  → ? ① Error
```

} index

Slicing string

- "Cut" part of a string

NOT INCLUDED

the index of the end of a subsequence (*not included*)

Syntax: [start : finish : step] — step: specifies the step size to jump along the sequence

start: the index of the start of a subsequence

*By default, these indices, (**start** and **finish**), will point to the **beginning** and **end** of the string, respectively.*

```
myStr = "Hello World"
print(myStr[1:6])   → 'ello '
print(myStr[1:2])   → 'e'
print(myStr[-7:-1]) → 'o Worl'
print(myStr[-3:-5]) → '' (empty)
print(myStr[:6])    → 'Hello '
print(myStr[5:])    → ' World'
```

Output

Characters	H	e	l	l	o		W	o	r	l	d
Indices	0	1	2	3	4	5	6	7	8	9	10
	−12	−10	−9	−8	−7	−6	−5	−4	−3	−2	−1

myStr[3:-2] = "lo wor"

If: Step = -ve ⇒ Start Undefined, Presume start LHS @ -1

Reversing str

```
aString = "Madam I'm Adam"
revString = aString[::-1]   → "madA m'I madaM"
```

Basic str operations

1) Length — Length of a string: `len()`
   e.g. `len(opStr)` → 5

2) Combine — Concatenate strings: `+`
   e.g. `opStr + "operations"` → 'Basic operations'

3) Repeat — Repeat String: `*`
   e.g. `opStr * 3` → 'BasicBasicBasic'

ASCII ⇒ unicode

ASCII	Unicode
• Uses 8 bits to store a character	• An extension of ASCII
• 2^8 = 256 different characters	• Able to include more characters
	• Uses 16 bits to store a character
	• 2^{16} = 65,536 characters

- The Unicode space is divided into 17 planes.
- Each plane contains 65,536 code points (16-bit).

*Total of: **1,114,112** characters, 96,000 used.*

From character → unicode: `ord('a')` → 97

From unicode → character: `chr(97)` → 'a'

① Used in encryption

② Used to compare char "a" < "b" ?

```
code = ord('a')
chr (code + 1)   → 'b'
```

IN

"contained within"

- Operator: `in`
- `a in b`: True if string **a** is contained in string **b**

```
myStr = "abcdefg"
'c' in myStr    → true
'cde' in myStr  → true
'cef' in myStr  → false
myStr in myStr  → true
```

Iterable: can use for loop to call

String.method()

Function - Code that "executes"

method - an "augmenter"
- dot notation invocation

Uppercase

```
myStr = "shouting!"
myStr.upper()  ⇒  'SHOUTING!'
```

Find

```
myStr = "Find in a string"
myStr.find('d')  ⇒  3
```

'd' is called an **argument** of the method.

If no char, output = -1

Join

```
str1 = "abcd"
str2 = "1234"
str2.join(str1)  ⇒  'a1234b1234c1234d'
str1.join(str2)  ⇒  '1abcd2abcd3abcd4'
```

`base.join(target)`

insert base into target

Mutable List!

Unlike str(" ")

Composite Type / Data Structure

① Built - in

1) String, list, Tuple, Dictionary

List

```
aList = list('abc')
  aList ⇒ ['a', 'b', 'c']

newList = [1, 3.14159, 'a', True]
```

Basic List operator

1) Addition — concatenate: + (only for lists – not string + list)
2) Repeat — repeat: *
3) Call for "n" — indexing: the [ ] operator), e.g., `lst[3]` ⇒ 4th item in the list
4) Cut — slicing: [:]
5) Check IN — membership: the `in` operator
6) Length — length: the `len()` function

Slicing List

myList	1	'a'	3.14159	True
Index Forward	0	1	2	3
Index Backward	–4	–3	–2	–1

```
myList[1]  ⇒  'a'

myList[:3]  ⇒  [1, 'a', 3.14159]
```

Indexing Lists

★ [5, X, "abc"][1] = X

List Operators

`min(lst)` Minimum element in the list

`max(lst)` Maximum element in the list

`sum(lst)` Sum of the elements, numeric only

```
myList[0] = 'a'      #index assignment
myList.append(e)     // e: element to append
myList.extend(L)     // L: a list
myList.pop(i)        // i: index (default: -1)
myList.insert(i,e)
myList.remove(e)
myList.sort()
myList.reverse()
```

```
myList = [1,3]                  [1, 3]
myList[0] = 'a'                 ['a', 3]
myList.append(2)                ['a', 3, 2]

lst = [6,5]
myList.extend(lst)              ['a', 3, 2, 6, 5]

myList.extend(5)                ERROR!

element = myList.pop()          ['a', 3, 2, 6]
print(element)                  5

myList.append([8,9])            ['a', 3, 2, 6, [8,9]]
```

```
myList.insert(0, 'b')           ['b', 'a', 3, 2, 6]
myList.insert(-1, 'b')          ['b', 'a', 3, 2, 'b', 6]
myList.insert(10, 'c')          ['b', 'a', 3, 2, 'b', 6, 'c']
myList.remove('b')              ['a', 3, 2, 'b', 6, 'c']

myList.sort()                   TypeError!!

myList.remove('b')              ['a', 3, 2, 6, 'c']
myList.remove('a')              [3, 2, 6, 'c']
myList.remove('c')              [3, 2, 6]

myList.remove('d')              ValueError!!

myList.sort()                   [2, 3, 6]

myList.reverse()                [6, 3, 2]
```

Split List

`.()` is what to split by

```
splitLst = 'this is a test'.split()
print(splitLst)  ⇒  ['this', 'is', 'a', 'test']
```

Join List

```
sortStr = ''.join(myLst)  ⇒  'abcxyz'
```

```
myLst = [4, 7, 1, 2]
myLst = myLst.sort()  ⇒ mylist.sort() is a null. Don't assign
myLst ⇒ None
```

```
[n**2 for n in range(1,6)]
```
→ [1, 4, 9, 16, 25]

```
[x + y for x in range(1,5) for y in range (1,4)]
```
→ ?

It is as if we had done the following:

```
myList = [ ]
for x in range (1,5):
  for y in range (1,4):
    myList.append(x+y)
```

```
[c for c in "Hi There Mom" if c.isupper()]
```
→ ['H', 'T', 'M']

Tuple

- Immutable List, can't be changed
- Everything works like list, except modify

Index / Slicing / len() / Print()

Creating Tuple

```
myTuple = 1,2        # creates (1,2)
myTuple = (1,)       # creates (1)
myTuple = (1)        # creates 1 not (1)
myTuple = 1,         # creates (1)
```

Dictionary

- two elements, a key & value
- Calling key returns the value

Access requires **[ ]** and the **key** is the index.

`myDict = {}` → **an empty dictionary**

`myDict['bill'] = 25` → **add the pair** `'bill':25`

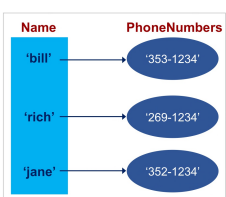
`print(myDict['bill'])` → **print 25**

`del myDic['bill']` → **remove the pair** `'bill':25`

{ } marker: used to create a dictionary

: marker: used to create **key:value** pairs

```
contacts = {'bill': '353-1234',
            'rich': '269-1234',
            'jane': '352-1234'}

print(contacts) → {'jane': '352-1234',
                   'bill': '353-1234',
                   'rich': '269-1234'}
```



Printing using FOR

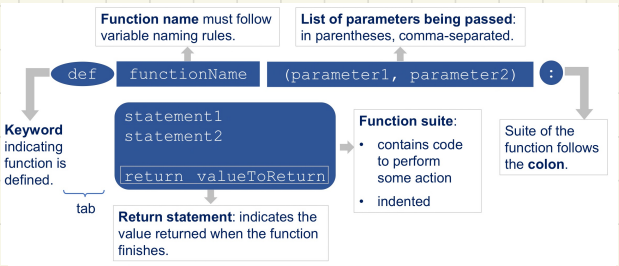
```
for key in myDict:
    print(key)            → prints all the keys

for key,value in myDict.items():
    print(key, value)     → prints all the key:value pairs

for value in myDict.values():
    print(value)          → prints all the values
```

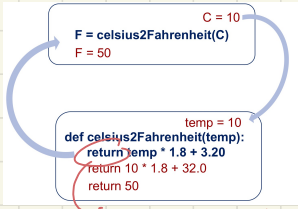
operators

- `len(myDict)` → **number** of **key:value pairs** in the dictionary
- `element in myDict` → boolean; is **element** a **key** in the dictionary?
- `for key in myDict` → iterate through the **keys** of a dictionary
- `myDict.items()` → return all the **key:value** pairs
- `myDict.keys()` → return all the keys
- `myDict.values()` → return all the values
- `myDict.clear()` → empty the dictionary
- `myDict.update(yourDict)` → for each key in `yourDict`, update `myDict` with that **key:value** pair

Function Development

- Input will give you Output
- for example, sqrt(10)





gives a value

Does one thing	Readable	Reusable	Complete	Not too Long
If it does too many things, it should be broken down into multiple functions (refactored).	If you write it, it should be readable. Give comments.	If it does one thing well, then when a similar situation (in another program) occurs, use it there as well.	A function should check for all the cases where it might be invoked. Check for potential errors.	Kind of synonymous with "**does one thing**". Use it as a measurement of doing too much.

If no return, it's a procedure

↳ Performs some duty, like printing or save file

If multiple return, end @ 1st return

```
def funcA (number):
      if number > 0:
            return "positive!"

      elif number < 0:
            return "negative!"

      else:
            return "zero!"
```

ABSTRACTION:

- View things @ different Levels of detail
  ↳ enough for info, not enough to be confused

Programs = Algorithms + Data Structures
↳ functions ↳ String, List

ABCD

A BCD A

AB CD BA

ABC D CBA

ABC DD CBA

Decomposition

- Break down problems into parts that are easier

1) Solve Complex problems
2) Collaboration & teamwork
3) Easy analysis

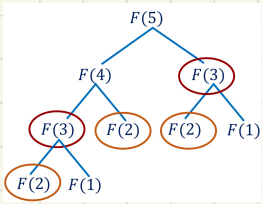
Divide & conquer

Recursive function

↳ Function that invokes itself

* May be redundant due to repeated computation

Look! Multiple calls of F(3), F(2) & F(1)



Base Case

Repeated step

```python
def reverser(a_str):
    if len(a_str) == 1:    # base case
        return a_str

    else:                  # recursive step
        new_str = reverser(a_str[1:])+ a_str[0]
        return new_str
```

Binary Tree

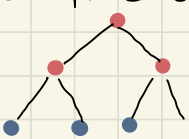
Root Node

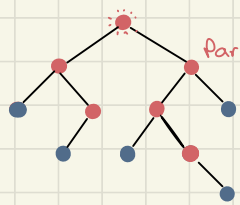
Parent Node

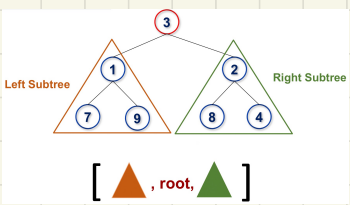
leaf node

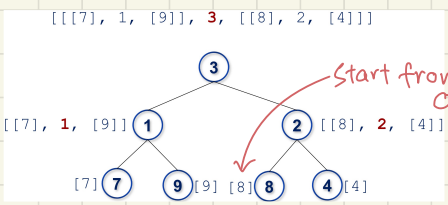
Complete Binary Tree

- Every node has Two Leafs





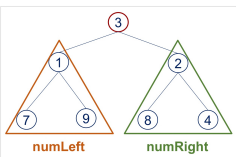




numOfNodes(t)	returns the total number of nodes in a CBT t
sumNodes(t)	returns the summation of all nodes in a CBT t
maxNode(t)	returns the maximum value of nodes in a CBT t
minNode(t)	returns the minimum value of nodes in a CBT t
mirror(t)	returns the mirrored CBT of a CBT t

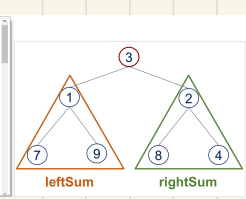
Number of nodes

```python
def numOfNodes(t):
    if len(t) == 1:
        return 1;

    else:
        numLeft = numOfNodes(t[0])

        numRight = numOfNodes(t[2])

        return ( numLeft + numRight + 1 )
```



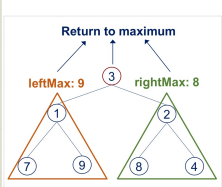
Sum of nodes

```python
def sumNodes(t):
    if len(t) == 1:
        return t[0];

    else:
        leftSum = sumNodes(t[0])

        rightSum = sumNodes(t[2])

        return ( t[1] + leftSum + rightSum)
```



Maximum Value In Tree

maxNode(t)



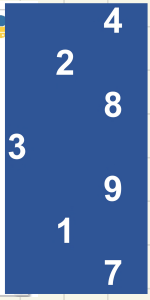

```python
def maxNode(t):
    if len(t) == 1:
        return t[0]
    else:
        leftMax = maxNode(t[0])
        rightMax = maxNode(t[2])

        maxValue = t[1]
        if leftMax > maxValue:
            maxValue = leftMax

        if rightMax > maxValue:
            maxValue = rightMax

        return maxValue
```

Mirror/Reflect Tree

```
def mirror(t):
    if len(t) == 1:
        return t

    else:
        parent = t[1]
        mirrorLTree = mirror(t[0])
        mirrorRTree = mirror(t[2])

        return [ mirrorRTree, parent, mirrorLTree ]
```

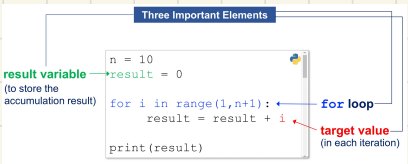
Print Tree

```
def printTree(t, level):
    if len(t) == 1:
        print("  " * level, end="")
        print(t[0])

    else:
        printTree(t[2], level + 1)

        print("  " * level, end="")
        print(t[1])

        printTree(t[0], level + 1)
```



Pattern Recognition

Iterative Accumulation $a+b+c+d=55$



0/1 - 1/2 + 2/3 - 3/4 + 4/5 - 5/6 + 6/7 - 7/8 + 8/9 - 9/10?

Three Important Elements

- result variable
- for loop
- target value

Iteration i

- i is odd: +(i-1)/i
- i is even: -(i-1)/i

```
n = 10
result = 0

for i in range(1,n+1):
    if i%2 == 1:
        result = result + (i-1)/i
    else:
        result = result - (i-1)/i

print(result)
```

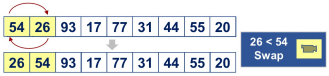
Sorting Items

Bubble

The bubble sort makes multiple passes through a list.

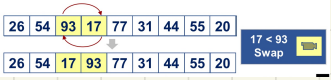
For each pass, the bubble sort is operated as follows:

1. Compare the first two number, and if the second number is smaller than the first, then the numbers are swapped.



2. Then move down one number, compare the number and the number that follows it, and swap the two numbers, if necessary.

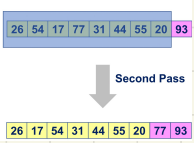
3. Repeat this process.



There are n-1 passes

At the **start of the second pass**,

- the largest value is now in place
- there are n – 1 items left to sort
- meaning, there will be n – 2 pairs



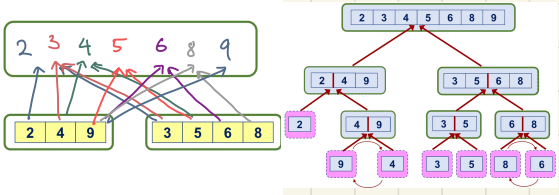
Merge Sort

Step 1 Compare the first elements of both lists one by one.

Step 2 Move the smaller element out of the list that it was found in. Add this value to the list of "sorted items".

Step 3 Repeat the process until only a single list remains.

Step 4 One list should still contain elements. This list is sorted. Move its contents into the result list.



Python Sorting

Python provides a module named operator that contains the useful itemgetter for sorting values other than the first and allows sorting on multiple values.

```
from operator import itemgetter
restaurant_info = [['Kentucky', 15, 6, 'Fried chicken'],
                   ['Macdonald', 12, 5,'Burger'],
                   ['Subway', 13, 7, 'Sandwiches']]

# filed 1 (index 0): name of restaurant
# field 2 (index 1): distance of restaurant
# field 3 (index 2): average price per person of the restaurant
# field 4 (index 3): signature dish of the restaurant

sort_info = sorted (restaurant_info, key = itemgetter(1))
print('sort by distance', sort_info)
```

Using TimSort
- Merge & insert sort

Bubble

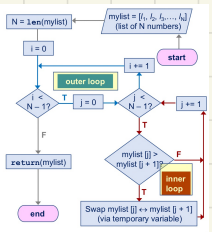
Step 1: Get the length of the sequence

Step 2: Take the first element and compare it with the immediate neighbor to the right: $a_i > a_{i+1}$
- If true: swap and increment *i* by one
- If false: increment *i* by one

Step 3: Repeat step 2, N – 1 times

Pseudocode

```
while k < N
        for i = 0 to N - 1
                if a[i] > a[i + 1]
                then
                        swap()
        end
        k = k + 1
end
```




```
def bubbleSort(alist):
    for passnum in range(len(alist)-1):
        for i in range(len(alist)-passnum-1):
            if alist[i]>alist[i+1]:
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp


alist = [54,26,93,17,77,31,44,55,20]
bubbleSort(alist)
print(alist)
```

Merge Sort

Step 1: If the input sequence has fewer than two elements, return.

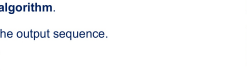
Step 2: Partition the input sequence into two halves.

Step 3: Sort the two subsequences **using the same algorithm**.

Step 4: Merge the two sorted subsequences to form the output sequence.

Pseudocode

```
MergeSort(list, first, last)
if (first < last)
        middle = (first + last) div 2
        MergeSort(list, first, middle)
        MergeSort(list, middle+1, last)
        Merge(list, first, middle, last)
end if
```



This means we have a list on the left and a list on the right.

Step 1 Compare the first elements of both lists one by one.

Step 2 Add this value to the list of "sorted items". Move the smaller element out of the list that it was found in.

Step 3 Repeat the process until only a single list remains.

Step 4 One list should still contain elements. This list is sorted. Move its contents into the result list.

Suggested code

```
while left_list and right_list:
    if left_list[0] < right_list[0]:
        result_list.append(left_list[0])
        left_list.pop(0)
    else:
        result_list.append(right_list[0])
        right_list.pop(0)
if left_list:
    result_list.extend(left_list)
else:
    result_list.extend(right_list)
```

```
def mergesort(list_of_items):
    list_len = len(list_of_items)

    # base case
    if list_len < 2:
        return list_of_items
    left_list = list_of_items[:list_len // 2]   # //
    right_list = list_of_items[list_len // 2:]  # "//" to force division

    # merge sort left and right list recursively
    left_list = mergesort(left_list)
    right_list = mergesort(right_list)
    return merge(left_list, right_list)
```

```
def merge(left_list, right_list):

    result_list = []

    # while left and right list has elements
    while left_list and right_list:
        if left_list[0] < right_list[0]:
            result_list.append(left_list[0])
            left_list.pop(0)
        else:
            result_list.append(right_list[0])
            right_list.pop(0)

    #left list still contain elements. Append its contents to end of the result
    if left_list:
        result_list.extend(left_list)
    else:
    #right list still contain elements. Append its contents to end of the result
        result_list.extend(right_list)

    return result_list
```

Search

index() provides where a key is located

iterative search	Binary search
- one by one	- for sorted lists
- Any list	- Divide by two, then look at the middle value.
	- Continously look @ middle value to find the result.