北京航空航天大學
BEIHANG UNIVERSITY

Lecture 4: Recursion

Xiaoqian Sun / Sebastian Wandelt

Beihang University

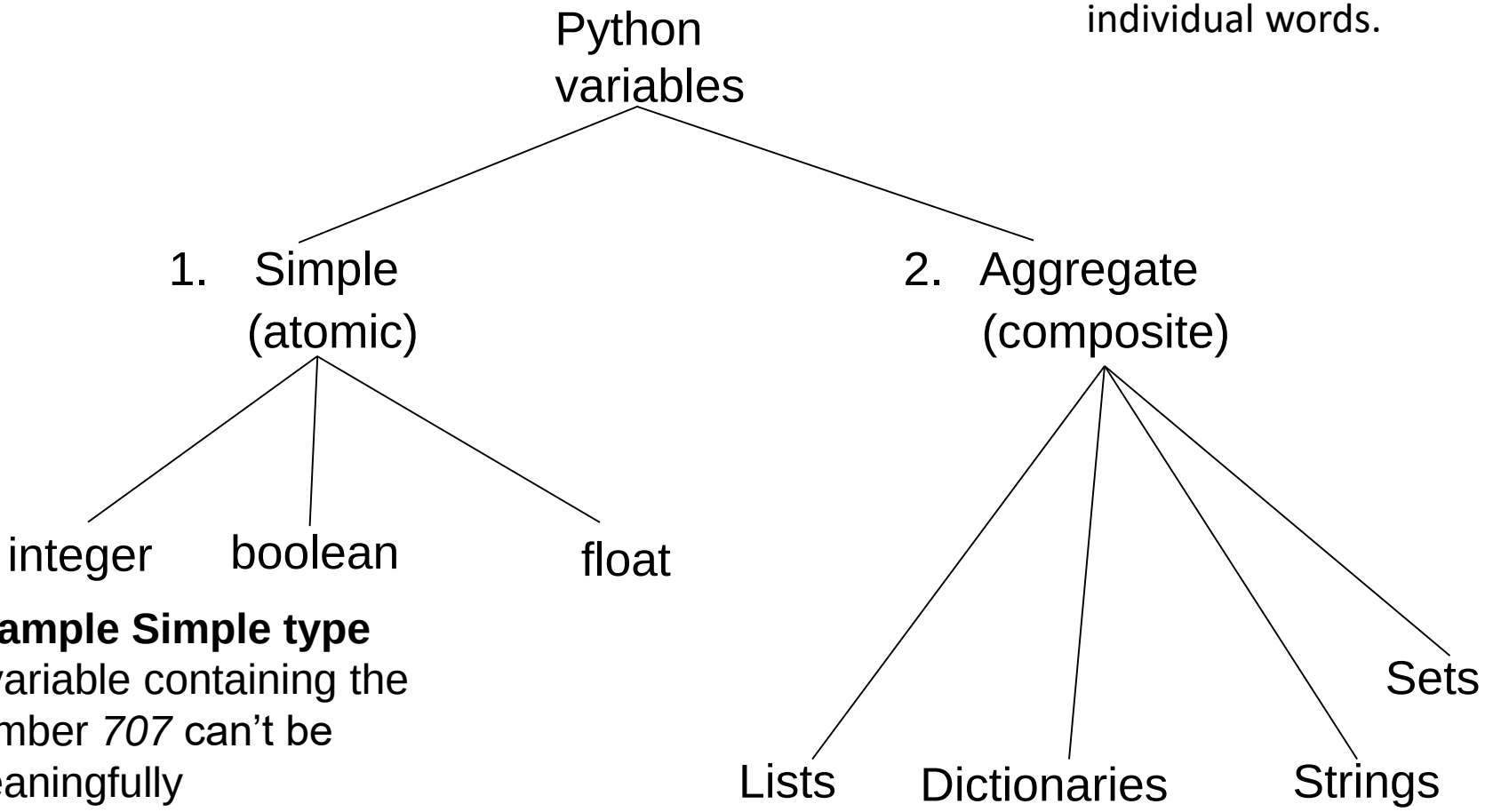
Outline

- Compound datatypes
- Recursion
- File management

Compound datatypes

Types Of Variables

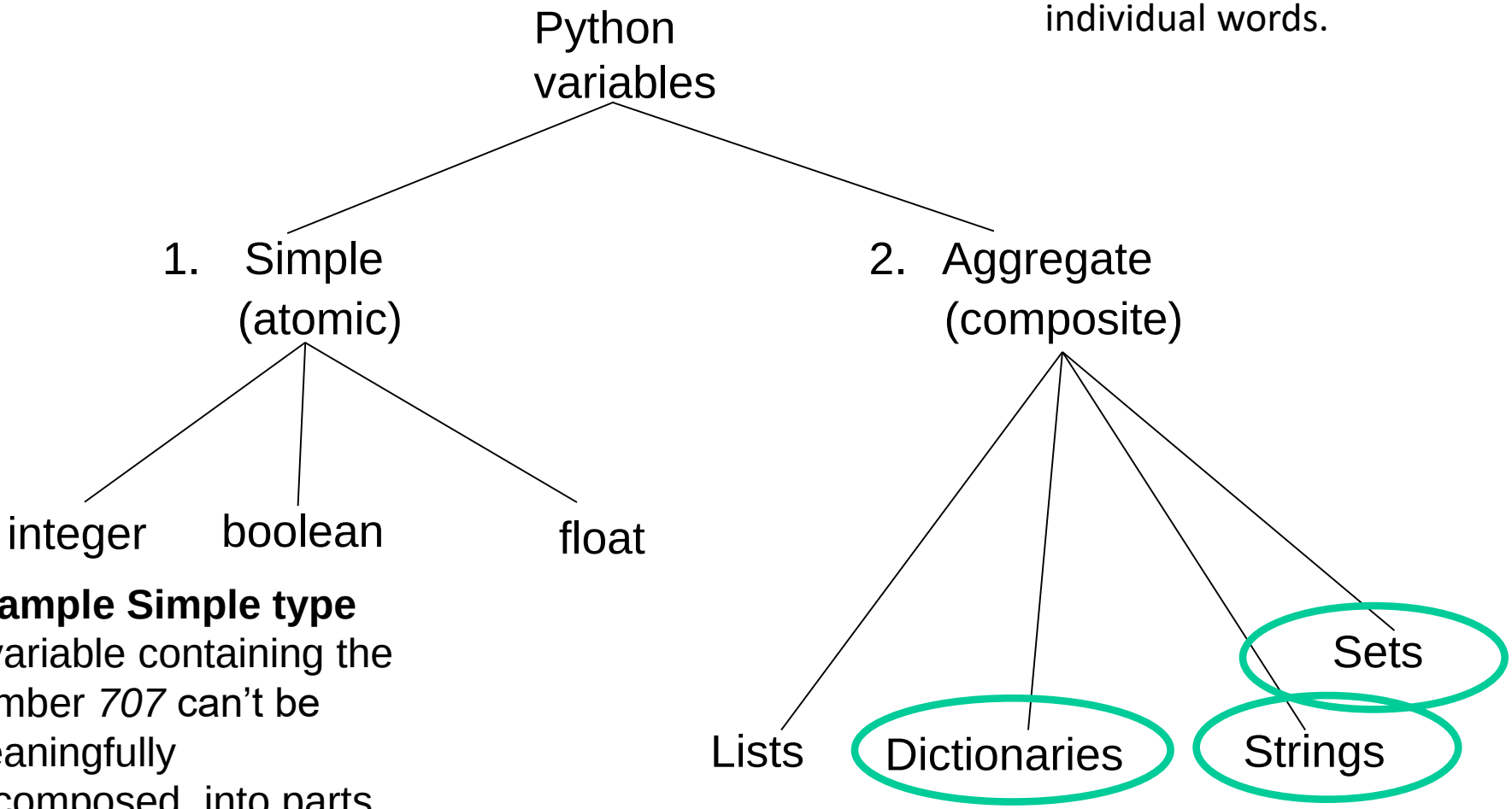
Example composite
A string (sequence of characters) can be decomposed into individual words.



Example Simple type
A variable containing the number *707* can't be meaningfully decomposed into parts

Types Of Variables

Example composite
A string (sequence of characters) can be decomposed into individual words.



Example Simple type
A variable containing the number *707* can't be meaningfully decomposed into parts

Compound datatypes
Strings

What are strings?

- Strings are a list of characters
- Almost all operations on lists also work on strings!

Compound datatypes
Dictionaries

What is a dictionary?

- You can think if it as a list of pairs, where the first element of the pair, the ***key***, is used to retrieve the second element, the ***value***.
- Thus we ***map a key to a value***
- Example:
  - Telephone book

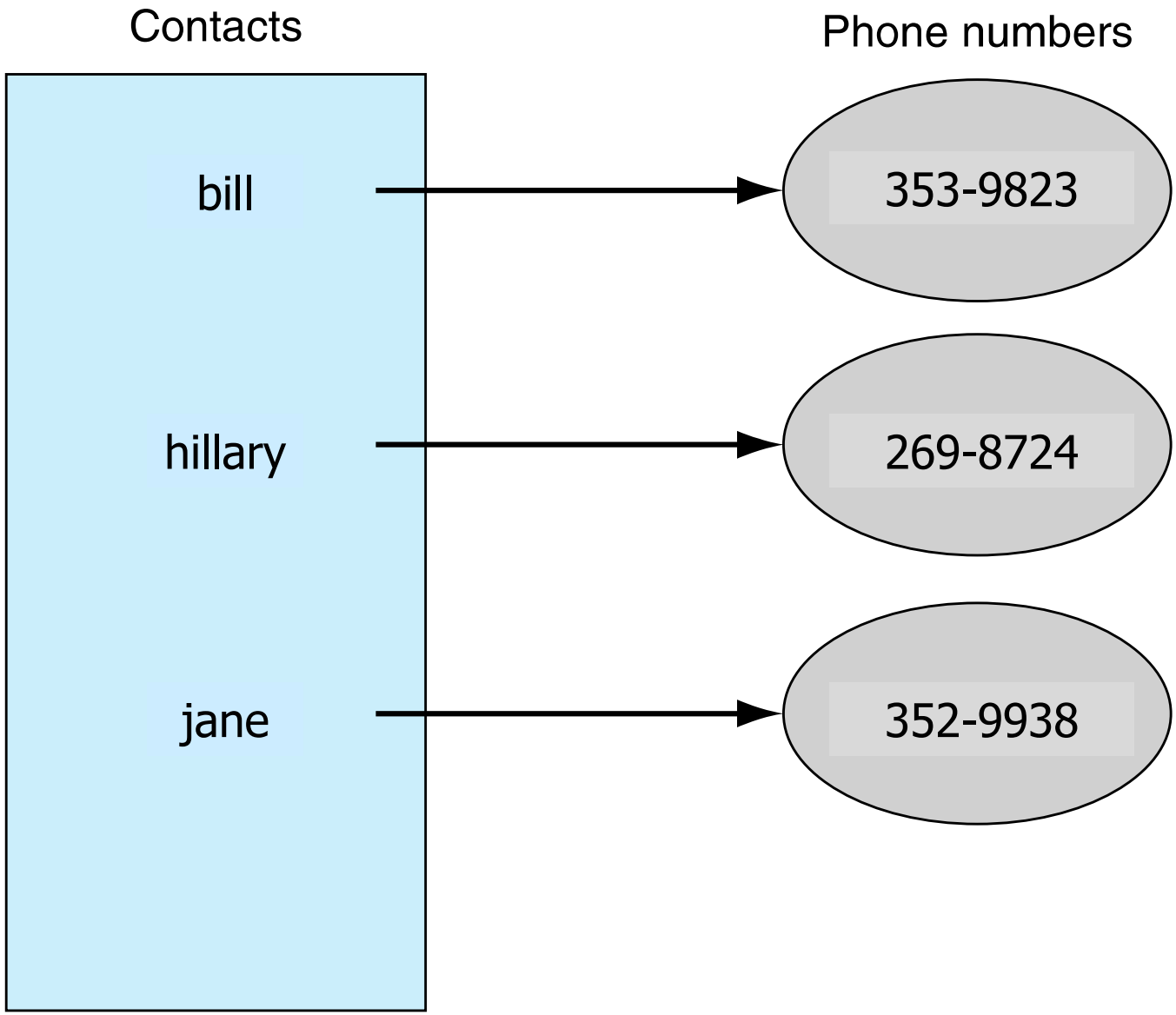
Key Value pairs

- The key acts as an index to find the associated value.
- Just like a dictionary, you look up a word by its spelling to find the associated definition
- A dictionary can be searched to locate the value associated with a key

Python Dictionary

- Use the { } marker to create a dictionary
- Use the : marker to indicate key:value pairs

```
contacts= {'bill':'353-9823',
 'hillary':'269-8724', 'jane':'352-9938'}
print (contacts)
{'jane':'352-9938',
 'bill':'353-9823',
 'hillary':'369-8724'}
```


Contacts
Phone numbers
bill
hillary
jane
353-9823
269-8724
352-9938

Keys and values

- Key must be immutable
  - strings, integers, tuples are fine
  - lists are NOT
- Value can be anything

Collections but not a sequence

- dictionaries are collections but they are not sequences such as lists, strings or tuples
  - there is no order to the elements of a dictionary
  - in fact, the order (for example, when printed) might change as elements are added or deleted.
- So how to access dictionary elements?

Access dictionary elements

Access requires `[ ]`, but the *key* is the index!

`my_dict={}`

- an empty dictionary

`my_dict['bill']=25`

- added the pair 'bill':25

`print(my_dict['bill'])`

- prints `25`

Dictionaries are mutable

- Like lists, dictionaries are a mutable data structure
  - you can change the object via various operations, such as index assignment

```
my_dict = {'bill':3, 'hillary':10}
print(my_dict['bill'])    # prints 3
my_dict['bill'] = 100
print(my_dict['bill'])    # prints 100
```

Common operators with Lists

Like others, dictionaries respond to these

- `len(my_dict)`
  - number of key:value **pairs** in the dictionary
- `element in my_dict`
  - boolean, is `element` a **key** in the dictionary
- `for key in my_dict:`
  - iterates through the **keys** of a dictionary

Dictionary content methods

- `my_dict.items()` – List of all the key/value pairs
- `my_dict.keys()` – List of all the keys
- `my_dict.values()` – List of all the values

How to print all values in a dictionary, each value in a separate row?

Example: Word frequencies

We are given a list of words

word_list=[‘she’,’dog’,cat’,’hello’,’he’,she’,’good’,’dog’]

and want to compute the word frequencies using a dictionary

Example: Word frequencies

```
count_dict = {}
for word in word_list:
    if word in count_dict:
        count_dict[word] += 1
    else:
        count_dict[word] = 1
```

Compound datatypes
Sets

Sets, as in Mathematical Sets

- in mathematics, a set is a collection of objects, potentially of many different types
- in a set, no two elements are identical. That is, a set consists of elements each of which is unique compared to the other elements
- there is no order to the elements of a set
- a set with no elements is the empty set

Creating a set

Set can be created in one of two ways:

- constructor:

```
my_set = set('abc')
my_set → {'a', 'b', 'c'}
```

- shortcut: { }

```
my_set = {'a', 'b','c'}
```

Diverse elements

- A set can consist of a mixture of different types of elements

```
my_set = {'a',1,3.14159,True}
```

- as long as the single argument can be iterated through, you can make a set of it

No duplicates

- duplicates are automatically removed

```
my_set = set("aabbccdd")
print(my_set)
```

→ `{'a', 'c', 'b', 'd'}`

Example

```
>>> null_set = set()              # set() creates the empty set
>>> null_set
set()
>>> a_set = {1,2,3,4}             # no colons means set
>>> a_set
{1, 2, 3, 4}
>>> b_set = {1,1,2,2,2}           # duplicates are ignored
>>> b_set
{1, 2}
>>> c_set = {'a', 1, 2.5, (5,6)} # different types is OK
>>> c_set
{(5, 6), 1, 2.5, 'a'}
>>> a_set = set("abcd")           # set constructed from iterable
>>> a_set
{'a', 'c', 'b', 'd'}              # order not maintained!
```

Common operators

Most data structures respond to these:

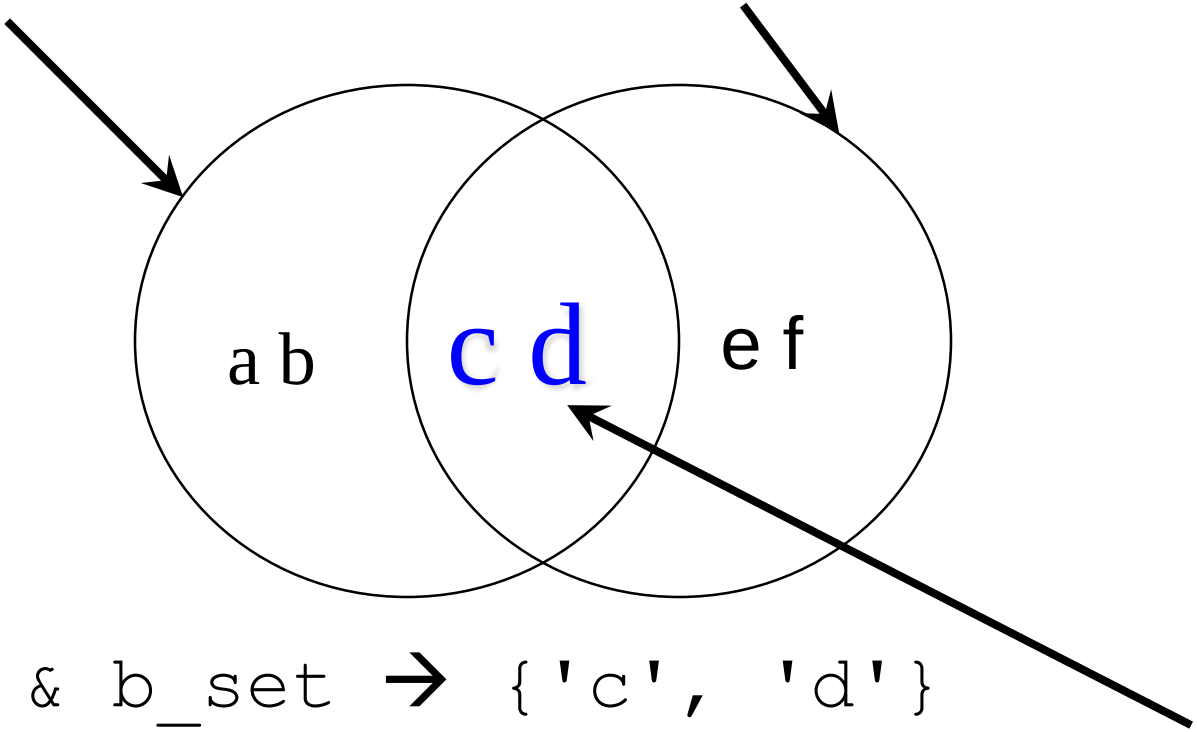
- `len(my_set)`
  - the number of elements in a set
- `element in my_set`
  - boolean indicating whether element is in the set
- `for element in my_set:`
  - iterate through the elements in `my_set`

Set operators

- The set data structure provides some special operators that correspond to the operators you learned in middle school.
- These are various combinations of set contents
- These operations have both a method name and a shortcut binary operator

method: intersection, op: &

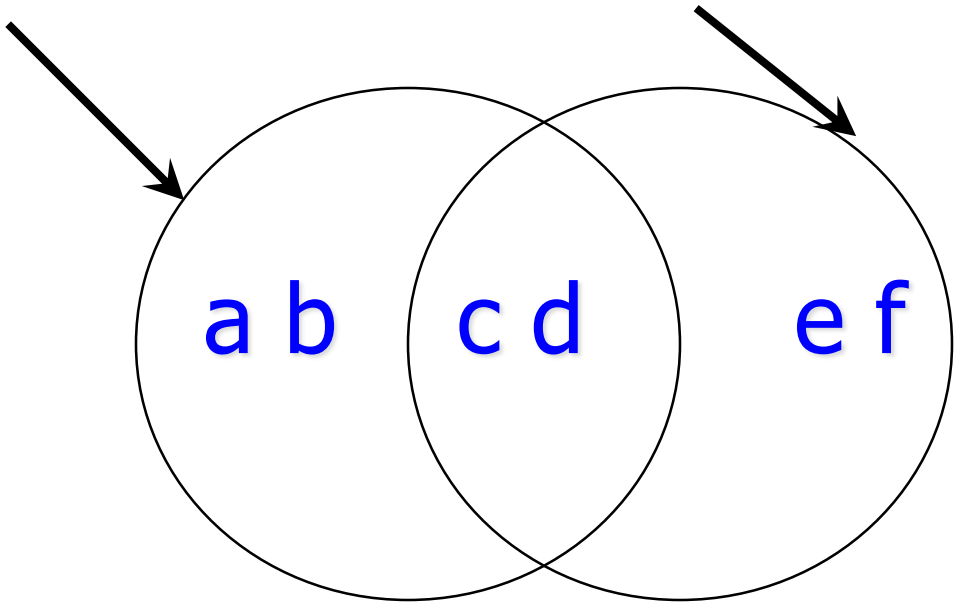
a_set=set("abcd") b_set=set("cdef")




```
a_set & b_set → {'c', 'd'}
b_set.intersection(a_set) → {'c', 'd'}
```

method: union, op: |

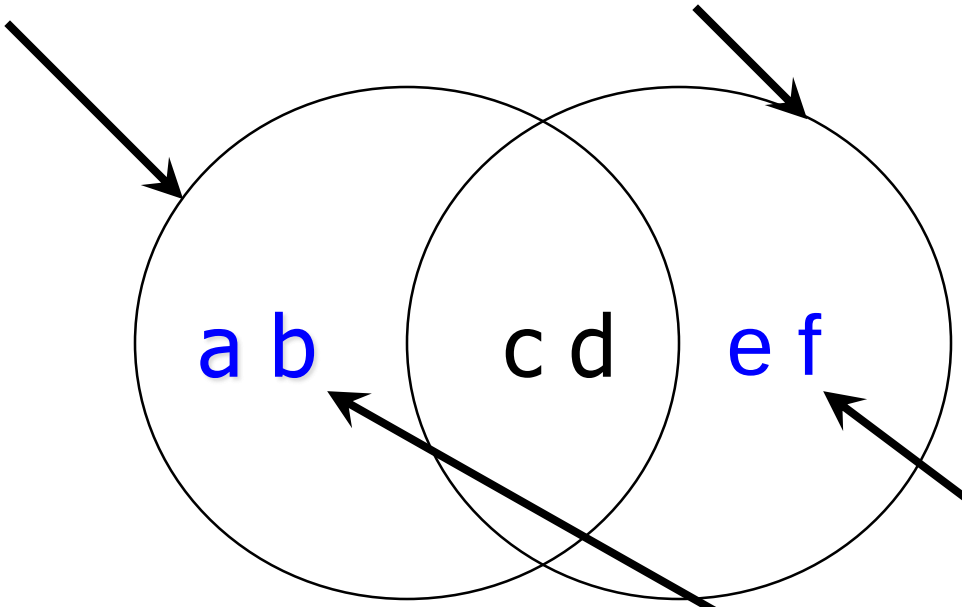
a_set=set("abcd") b_set=set("cdef")




```
a_set | b_set → {'a', 'b', 'c', 'd', 'e', 'f'}
b_set.union(a_set) → {'a', 'b', 'c', 'd', 'e',
'f'}
```

method:difference op: -

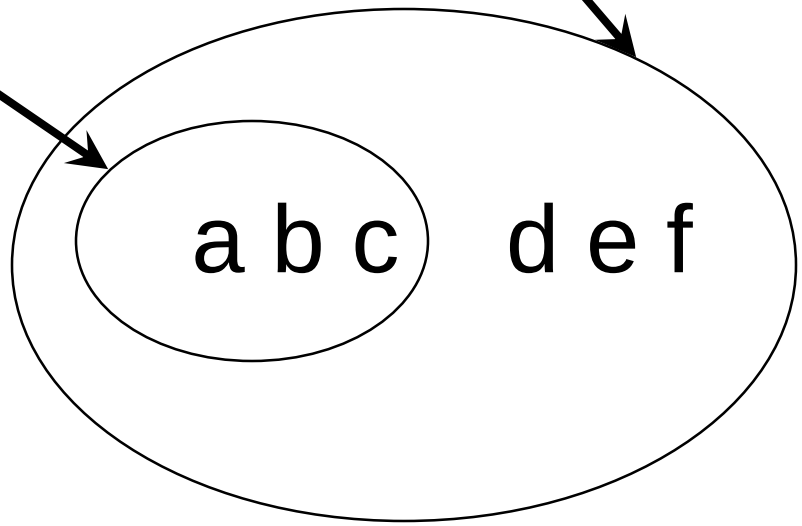
a_set=set("abcd") b_set=set("cdef")




```
a_set - b_set → {'a', 'b'}
b_set.difference(a_set) → {'e', 'f'}
```

method: issubset, op: <=
method: issuperset, op: >=

small_set=set("abc"); big_set=set("abcdef")




```
small_set <= big_set → True
big_set >= small_set → True
```

Other Set Ops

- `my_set.add("g")`
  - adds to the set, no effect if item is in set already
- `my_set.clear()`
  - empties the set
- `my_set.remove("g")` versus `my_set.discard("g")`
  - `remove` throws an error if "`g`" isn't there. `discard` doesn't care. Both remove "`g`" from the set
- `my_set.copy()`
  - returns a shallow copy of `my_set`

Recursion

What Is Recursion?

"*the determination of a succession of elements by operation on one or more preceding elements according to a rule or formula involving a finite number of steps*" (Merriam-Webster online)

What This Really Means

Breaking a problem down into a sequence of steps. The final step is reached when some basic condition is satisfied.

The solution for each step is used to solve the previous step.

The solution for all the steps together form the solution to the whole problem.

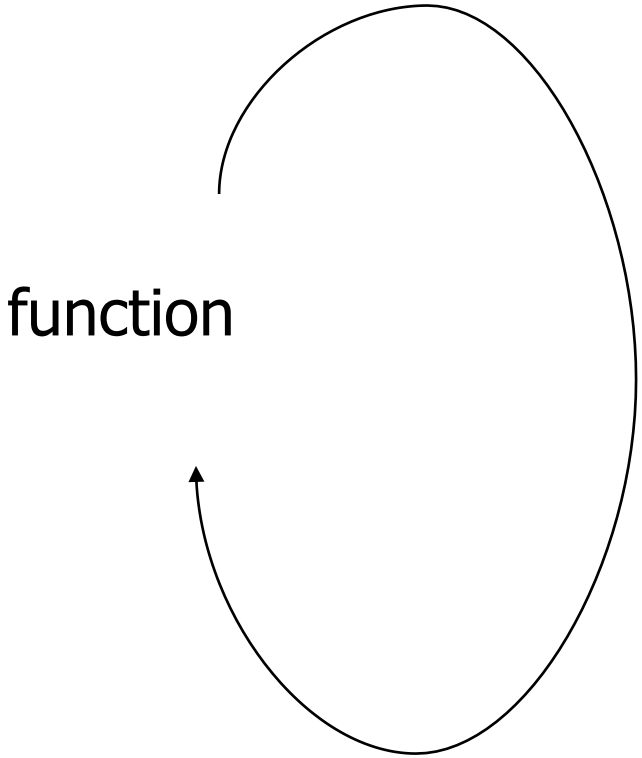
Recursion In Real Life?

Do you have an example for recursion in real life?

Recursion In Programming

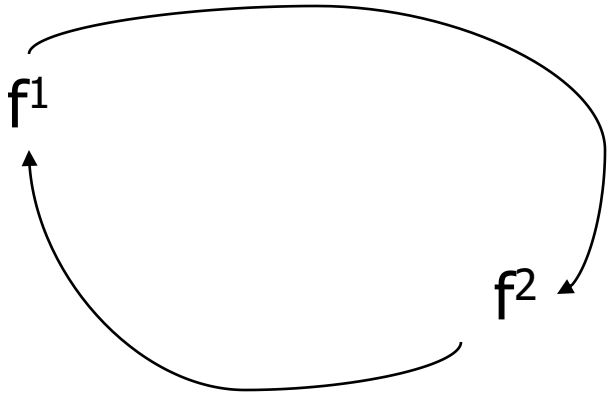
"*A programming technique whereby a function calls itself either directly or indirectly*."

Direct Call

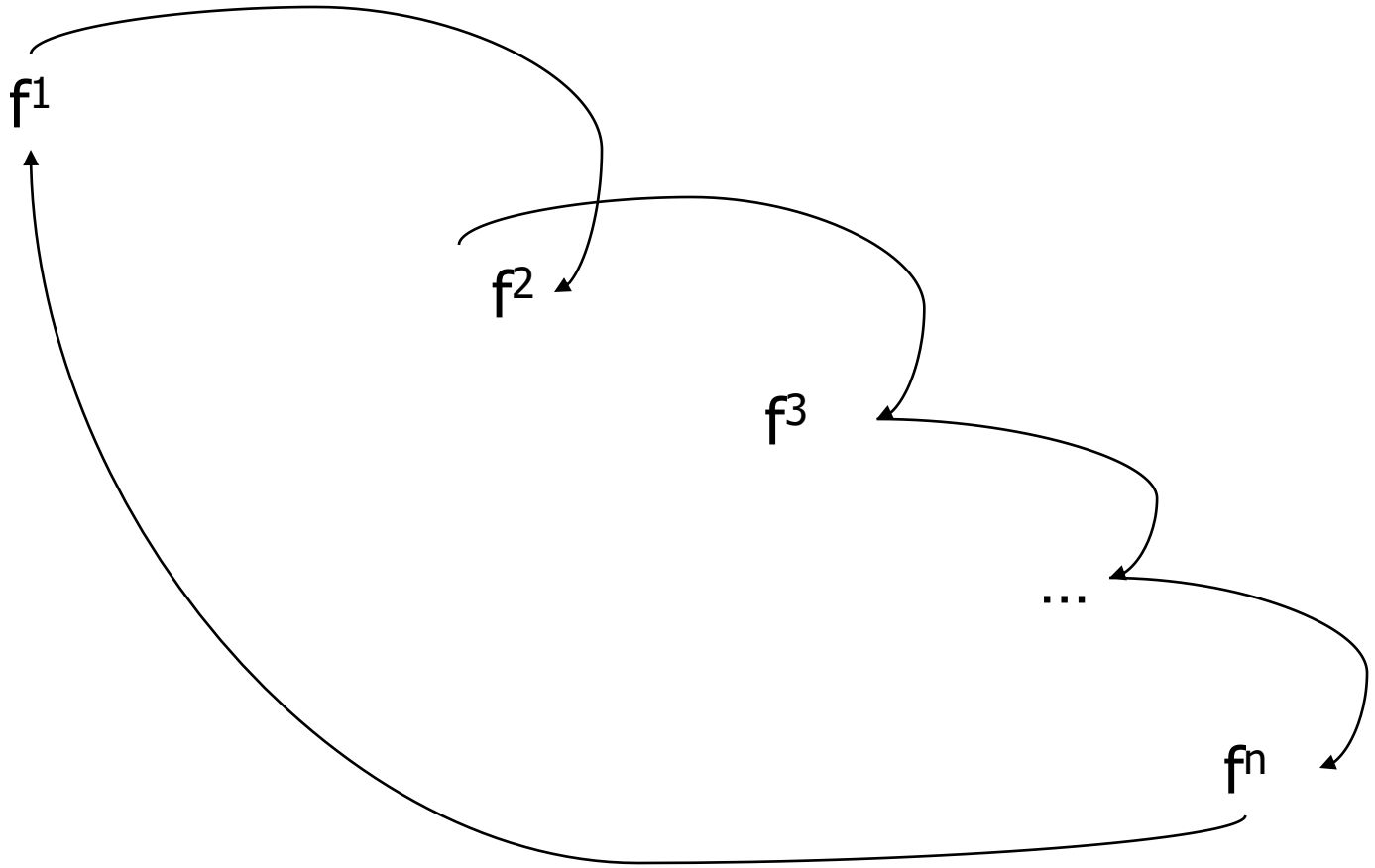



```
def fun ():
    …
  fun ()
    …
```

Indirect Call



Indirect Call



Indirect Call (2)

Example for indirect call:

```
def fun1():
   fun2()

def fun2():
   fun1()

fun1()
```

Requirements For *Sensible* Recursion

1) Base case (similar to termination criterion for loops)
2) Progress is made (towards the base case)

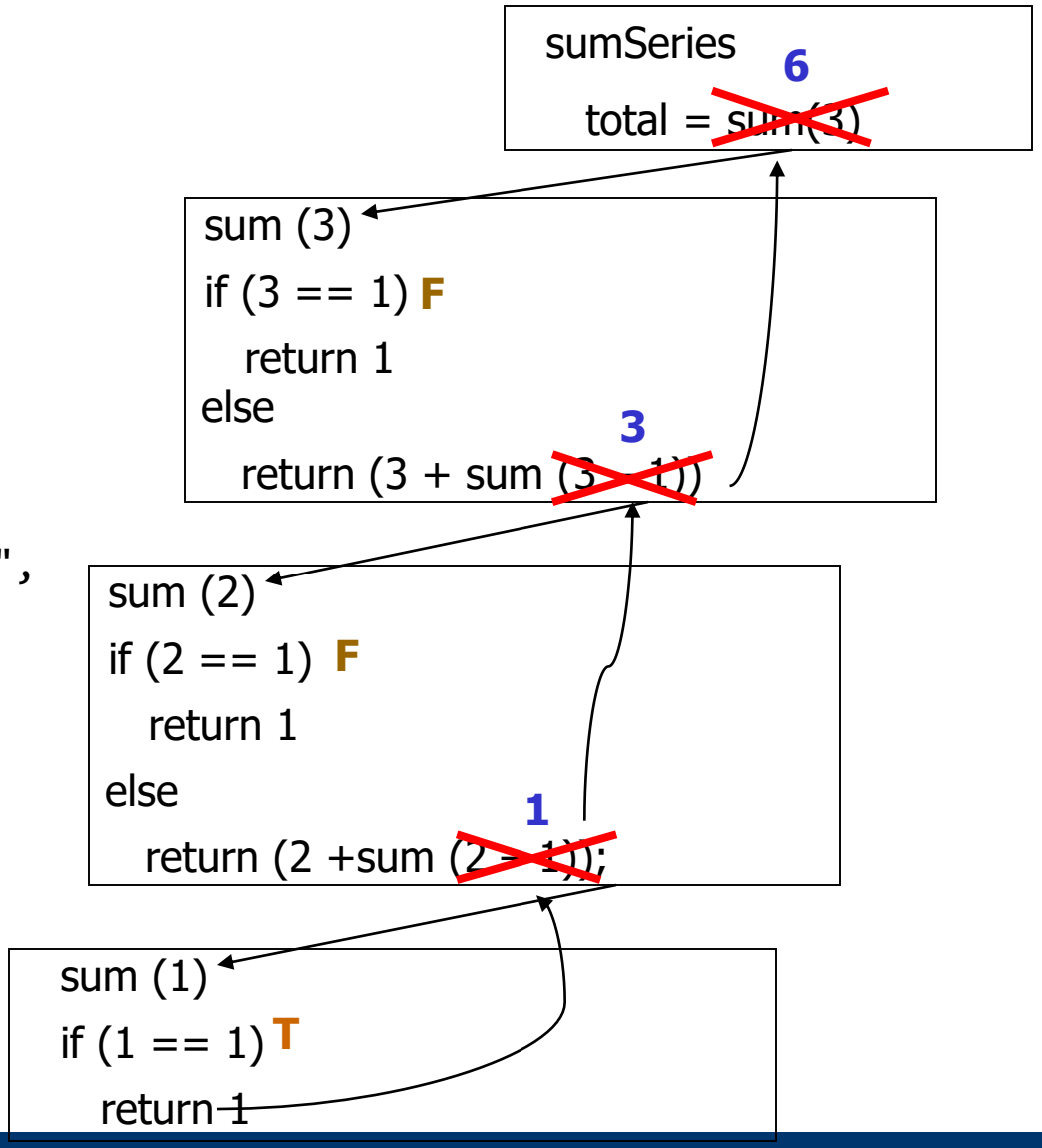
Example: Summing up numbers 1 to N

Easy to do with loops (see Exercise session).

How to achieve this with recursion?

Example Program: sumSeries.py

```
def sum(no):
    if (no == 1):
        return 1
    else:
        return (no + sum(no-1) )


def start():
    last = input ("Enter the last
                   number: ")
    last = (int)last
    total = sum(last)
    print ("The sum of the series
           from 1 to", last, "is",
           total)

start()
```



When To Use Recursion

- When a problem can be divided into steps.
- The result of one step can be used in a previous step.
- There is a scenario when you can stop sub-dividing the problem into steps (step = recursive call) and return to a previous step.
  - Algorithm goes back to previous step with a partial solution to the problem (back tracking)
- All of the results together solve the problem.

When To Consider Alternatives To Recursion

- When a loop will solve the problem just as well
- Types of recursion (for both types a `return` statement is excepted)
  - **Tail recursion**
    - The last statement in the function is another recursive call to that function This form of recursion can easily be replaced with a loop.
  - **Non-tail recursion**
    - The last statement in the recursive function is not a recursive call.
    - This form of recursion is very difficult (read: impossible) to replace with a loop.

Second Example for Recursion

- Consider the mathematical definition of b^n, where n is a non-negative integer:

  $b^n = 1$ if $n = 0$

  $b^n = b \times b^{n-1}$ if $n > 0$

- This definition is recursive, and immediately suggests a recursive function:

```
def power (b, n):
    if n == 0:
        return 1
    else:
        return b * power(b, n-1)
```

Error Handling Example Using Recursion

- Name of the example program: `errorHandling.py`

  – Iterative/looping solution (month must be between 1 – 12)

```
month = -1
while ((month < 1) or (month > 12)):
        month = int(input("Enter birth month (1-12): "))
```

Error Handling Example Using Recursion (2)

- Iterative/looping solution (day must be between 1 – 31)

```
def promptDay():
    day = int(input("Enter day of birth (1-31): "))
    if ((day < 1) or (day > 31)):
        day = promptDay()
    return(day)

...
day = promptDay()
```

Drawbacks Of Recursion

Function calls can be costly

- Uses up memory
- Uses up time

Benefits Of Using Recursion

- Simpler solution that's more elegant (for many standard problems)
- Easier to visualize solutions (for some people and certain classes of problems – typically require either: non-tail recursion to be implemented or some form of "backtracking")

Common Pitfalls When Using Recursion

- These three pitfalls can result in a runtime error
  - No base case
  - No progress towards the base case
  - Using up too many resources (e.g., variable declarations) for each function call

No Base Case

```
def sum(no):
      return(no + sum (no - 1))
```

No Base Case

```
def sum (no):
    return (no + sum (no - 1))
```

No Progress Towards The Base Case

```
def sum (no):
    if (no == 1):
        return 1
    else:
        return (no + sum (no))
```

No Progress Towards The Base Case

```
def sum (no):
    if (no == 1):
        return 1
    else:
        return (no + sum (no))
```

The recursive case doesn't make any progress towards the base (stopping) case

Using Up Too Many Resources

- Name of the example program: `recursiveBloat.py`

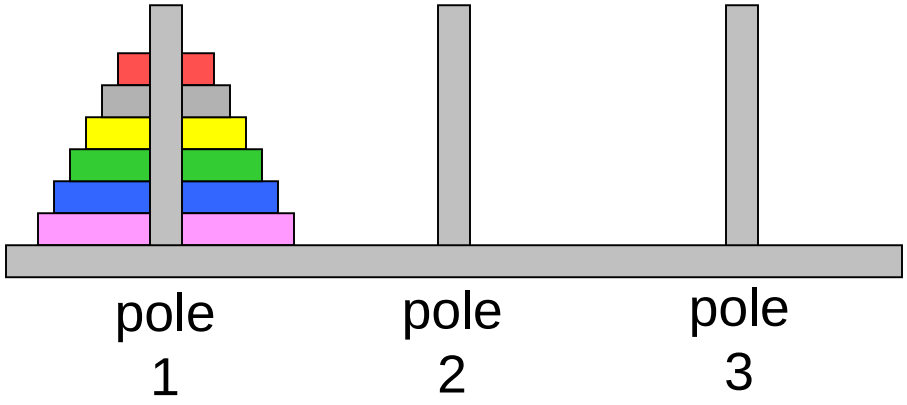
```
def fun(no):
    print(no)
    aList = []
    for i in range (0, 10000000, 1):
       aList.append("*")
    no = no + 1
    fun(no)

fun(1)
```

You Should Now Know

- What is a recursive computer program
- How to write and trace simple recursive programs
- What are the requirements for recursion/What are the common pitfalls of recursion

Larger example: Towers of Hanoi *(1)*

- Three vertical poles are mounted on a platform.
- A number of discs are provided, all with different diameters. Each disc has a hole in its centre.
- All discs are initially threaded on to pole 1, forming a tower with the largest disc at the bottom and the smallest disc at the top.

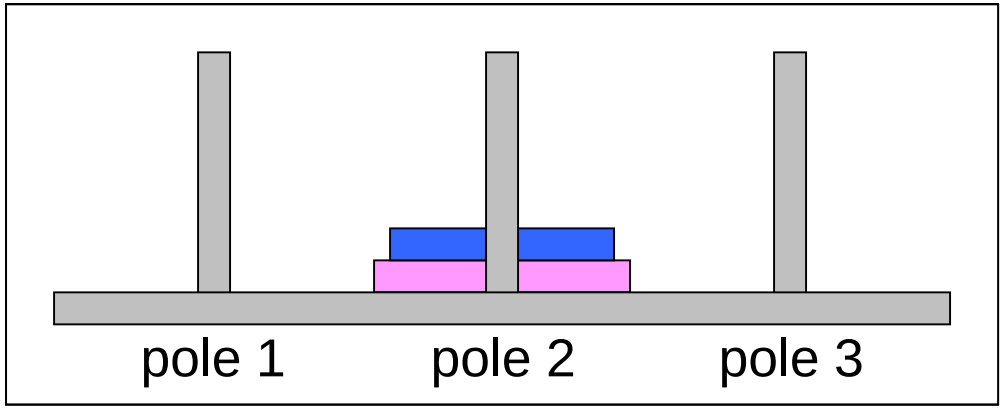


Example: Towers of Hanoi *(2)*

- *One* disc may be moved at a time, from the top of one pole to the top of another pole.
- A larger disc may not be moved on top of a smaller disc.
- Problem: Move the tower of discs from pole 1 to pole 2.

Example: Towers of Hanoi *(3)*

- Animation (with 2 discs):



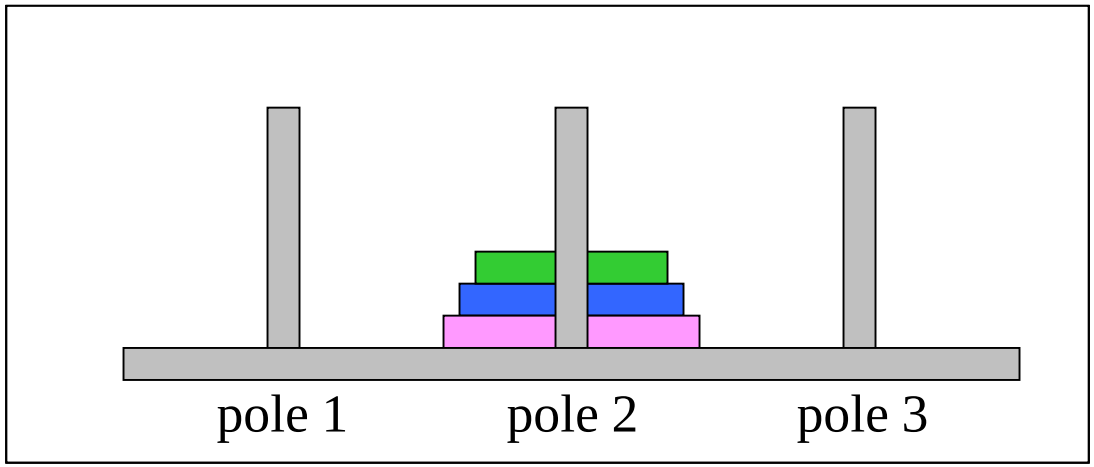
How to implement a solution algorithm?

- We just want to have a printout of disk movement instructions for a given number of disks n
  - Example:

```
Move disc from 1 to 2.
Move disc from 1 to 3.
Move disc from 2 to 3.
Move disc from 1 to 2.
Move disc from 3 to 1.
Move disc from 3 to 2.
Move disc from 1 to 2.
```

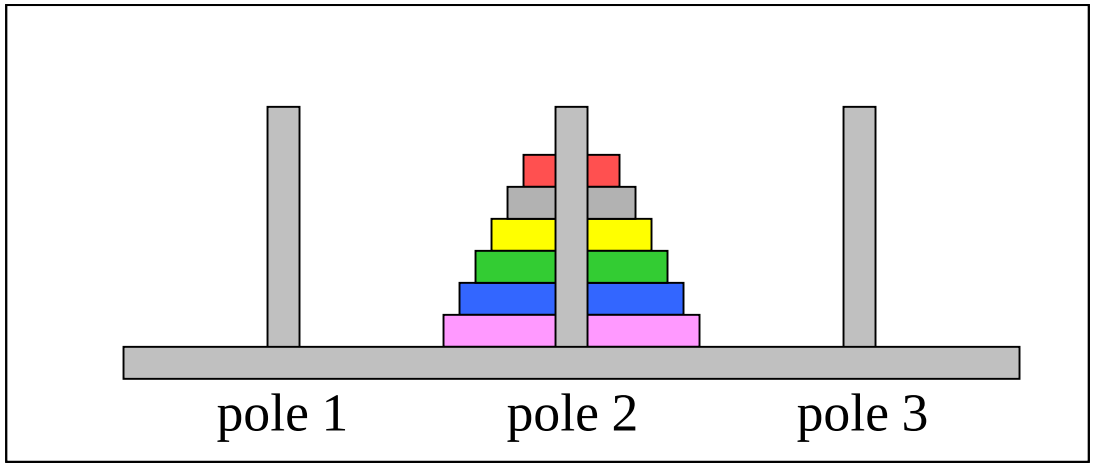
Example: Towers of Hanoi *(6)*

- Animation (with 3 discs):



Example: Towers of Hanoi *(7)*

- Animation (with 6 discs):



Example: Towers of Hanoi *(4)*

- Implementation:

```
def hanoi (n):
    # Move a tower of n discs from tower 1 to tower 2.
    move_tower(n, 1, 2, 3)
def move_tower (n, a, b, c):
    # Move a tower of n discs from the top of tower a
    # to the top of tower b, using tower c as a spare.
    if n == 1:
        move_disc(a, b)
    else:
        move_tower(n-1, a, c, b)
        move_disc(a, b)
        move_tower(n-1, c, b, a)
```

Example: Towers of Hanoi *(5)*

- Implementation *(continued)*:

```
def move_disc (a, b):
    # Move a single disc from tower a to tower b.
    print 'Move disc from %d to %d.' \
        % (a, b)
```

- Output (with 3 discs):

```
Move disc from 1 to 2.
Move disc from 1 to 3.
Move disc from 2 to 3.
Move disc from 1 to 2.
Move disc from 3 to 1.
Move disc from 3 to 2.
Move disc from 1 to 2.
```

Files access in Python

What You Need In Order To Read Information From A File

1. Open the file and associate the file with a file variable.
2. A command to read the information.
3. A command to close the file.

1. Opening Files

Prepares the file for reading:

A. Links the file variable with the physical file (references to the file variable are references to the physical file).
B. Positions the file pointer at the start of the file.

Format:[1]

```
<file variable> = open(<file name>, "r")
```

Example:

(Constant file name)

```
inputFile = open("data.txt", "r")
```

OR

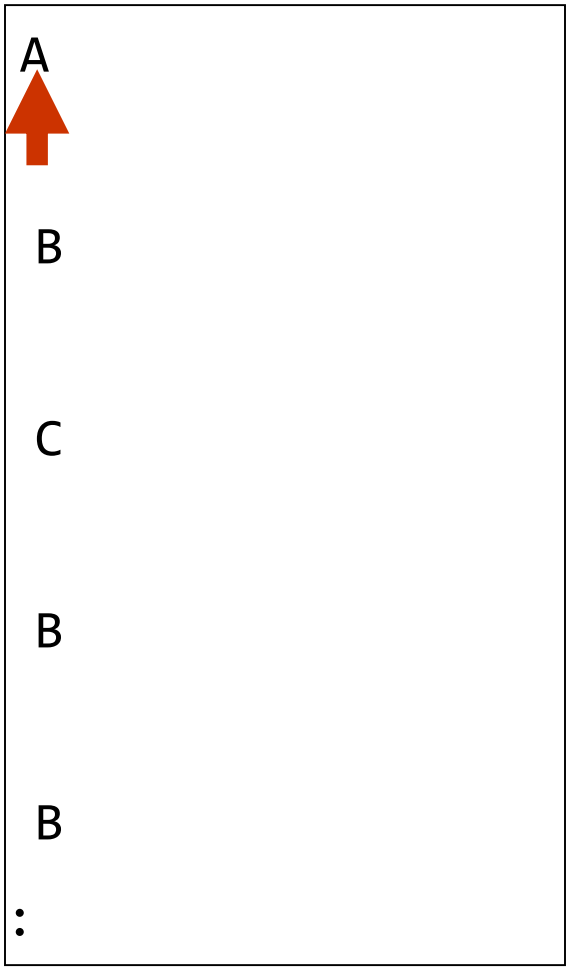
(Variable file name: entered by user at runtime)

```
filename = input("Enter name of input file: ")
inputFile = open(filename, "r")
```

1 Examples assume that the file is in the same directory/folder as the Python program.

B. Positioning The File Pointer

letters.txt



2. Reading Information From Files

- Typically reading is done within the body of a loop
- Each execution of the loop will read a line from file into a string

Format:

```
for <variable to store a string> in <name of file variable>:
    <Do something with the string read from file>
```

Example:

```
for line in inputFile:
    print(line)  # Echo file contents back onscreen
```

Closing The File

- Although a file is automatically closed when your program ends it is still a good style to explicitly close your file as soon as the program is done with it.
  - What if the program encounters a runtime error and crashes before it reaches the end? The input file may remain 'locked' an inaccessible state because it's still open.
- **Format:**

  `<name of file variable>.<close>()`

- **Example:**

```
inputFile.close()
```

Reading From Files: Putting It All Together

Name of the online example: `grades1.py`

Input files: `letters.txt or gpa.txt`

```
inputFileName = input("Enter name of input file: ")
inputFile = open(inputFileName, "r")
print("Opening file", inputFileName, " for reading.")

for line in inputFile:
    sys.stdout.write(line)


inputFile.close()
print("Completed reading of file", inputFileName)
```

What You Need To Write Information To A File

1. Open the file and associate the file with a file variable (file is "locked" for writing).
2. A command to write the information.
3. A command to close the file.

1. Opening The File

Format[1]:

```
<name of file variable> = open(<file name>, "w")
```

Example:

(Constant file name)

```
outputFile = open("gpa.txt", "w")
```

(Variable file name: entered by user at runtime)

```
outputFileName = input("Enter the name of the output file
                        to record the GPA's to: ")
outputFile = open(outputFileName, "w")
```

1 Typically the file is created in the same directory/folder as the Python program.

3. Writing To A File

- You can use the '`write()`' function in conjunction with a file variable.
- Note however that this function will ONLY take a string parameter (everything else must be converted to this type first).

Format:

```
outputFile.write(temp)
```

Example:

```
# Assume that temp contains a string of characters.
outputFile.write (temp)
```

Data Processing: Files

- Files can be used to store complex data given that there exists a predefined format.
- Format of the example input file: ‘`employees.txt`’

  <Last name><SP>*<First Name>*,*<Occupation>*,*<Income>*

Example Program: data_processing.py

```
# EMPLOYEES.TXT
Adama Lee,CAG,30000
Morris Heather,Heroine,0
Lee Bruce,JKD master,100000
```

```
inputFile = open ("employees.txt", "r")

print ("Reading from file input.txt")
for line in inputFile:
    name,job,income = line.split(',')  # Divided by the comma
    first,last = name.split()
    income = int(income)
    income = income + (income * BONUS)
    print("Name: %s, %s\t\t\tJob: %s\t\tIncome $%.2f"
            %(first,last,job,income))

print ("Completed reading of file input.txt")
inputFile.close()
```

Error Handling With Exceptions

- Exceptions are used to deal with extraordinary errors ('exceptional ones').
- Typically these are fatal runtime errors ("crashes" program)
- Example: trying to open a non-existent file
- Basic structure of handling exceptions

```
Try:
    Attempt something where exception error may happen
Except:
    React to the error
Else:  # Not always needed
    What to do if no error is encountered
Finally:  # Not always needed
    Actions that must always be performed
```

Exceptions: File Example

- Name of the online example: `file_exception.py`
- Input file name: Most of the previous input files can be used e.g. "`input1.txt`"

```
inputFileOK = False
while (inputFileOK == False):
     try:
         inputFileName = input("Enter name of input file: ")
         inputFile = open(inputFileName, "r")
         print("Opening file" + inputFileName, " for
                reading.")
         inputFileOK = True
      for line in inputFile:
         sys.stdout.write(line)
      print("Completed reading of file", inputFileName)
```

Exceptions: File Example (2)

```
    # Still inside the body of the while loop (continued)
            inputFile.close()
            print("Closed file", inputFileName) # End of try-body
        except IOError:
            print("Error: File", inputFileName, "could not be
                  opened")
        else:
            print("Successfully read information from file",
                   inputFileName)
        finally:
            print("Finished file input and output\n")
```

What you should know after this lecture ..

- The concepts of compound data types
  - Strings, Lists, Sets, Dictionaries
- What is recursion and how it works
- How to open a file for reading/writing

Thank you very much!