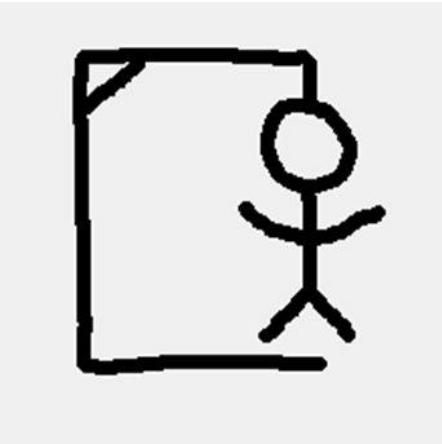


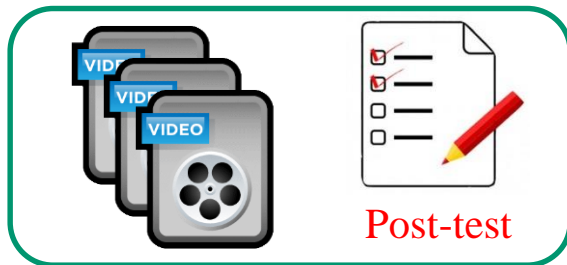


# CMPUT 175

## Introduction to Foundations of Computing



Simple enciphering and  
Hangman Word Game  
Design and Implementation  
with Python



You should view the vignettes:  
Transposition Cipher  
Need for Data Structures

# Objectives

- Revision of Programming concepts learned in CMPUT 174
- Review of Basic Python from CMPUT 174
- Discussing how to solve a problem to be implemented in a programming language
- Implementing simple enciphering and Hangman word game in Python
- Maybe learn something new
- Have fun!

# Simple Enciphering (Deciphering)

rrd ehtHa#tcao sd ea#oe crihewg#Mhe mf ahsm#  
yeinpobyaoe# rveuri.sms#b ewt r e.#

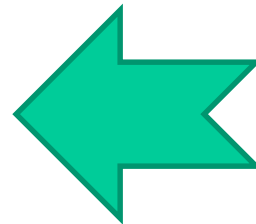
Private key is “CMPT175”

**Method:** get order of letters in Private key

	0	1	2	3	4	5	6
→	1	5	7	C	M	P	T
	0	1	2	3	4	5	6
	C	M	P	T	1	7	5
→	3	4	5	6	0	2	1

My brot  
her rec  
e ived a  
new co  
mputer  
for his  
birthd  
a y. He  
has awe  
some ga  
mes .

My brother received a new  
computer for his birthday.  
He has awesome games.



# Simple Enciphering (Enciphering)

My brother received a new computer for his birthday. He has awesome games.

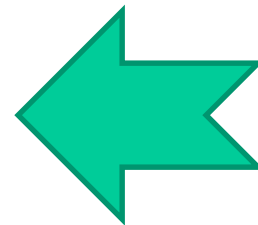
Private key is “CMPT175”

**Method:** get order of letters in Private key

	0	1	2	3	4	5	6
→	1	5	7	C	M	P	T
	C	M	P	T	1	7	5
→	3	4	5	6	0	2	1

My brot  
her rec  
eived a  
new co  
mputer  
for his  
birthd  
ay. He  
has awe  
some ga  
mes.

rrd ehtHa#tcao sd ea#  
oe crihewg#Mhe mf ahsm#  
yeinpobyaoe# rveuri.sms#  
b ewt r e.#



Slice the message  
by column in order

# Simple Enciphering

- Encryption

- Input

Message + Private Key

- Data processing

Do the trick to manipulate the string

- Output

Encrypted Message

---

- Decryption

- Input

Encrypted Message + Private Key

- Data processing

Do the reverse trick to manipulate the string

- Output

Clear Message

# Simple Enciphering

- We need 2 strings one for the clear messages and one for the enciphered message

```
myMessage = "My brother received a new computer for his birthday. He has awesome games."
```

```
Cipher = "rrd ehtHa#tcao sd ea#oe crihewg#Mhe mf ahsm#yeinpobyaoe# rveuri.sms#b ewt r e. #"
```

- We need a string for the private key `myPrivateKey = "CMPT175"`
- We need a list for the ordering of the letters in the key

```
myOrder = [3,4,5,6,0,2,1]
```

- We need a list of strings for the snips
- And some other containers for temporary variables for processing

```
[ "My brot",  
  "her rec",  
  "eived a",  
  " new co",  
  "mputer ",  
  "for his",  
  " birthd",  
  "ay. He ",  
  "has awe",  
  "some ga",  
  "mes. " ]
```

# Simple Enciphering Flow Chart

## 1-Read Message and PrvKey

```
myMessage = "My brother received a new computer for his birthday. He has awesome games."  
myPrivateKey = "CMPT175"
```

## 2-Get order of letters in PrvKey

```
myOrder = [3,4,5,6,0,2,1]
```

### 3-Cut the Message in snips the size of the PrvKey and align them

```
["My brot",
"her rec",
"eived a",
" new co",
"mputer ",
"for his",
" birthd",
"ay. He ",
"has awe",
"some ga",
"mes."]
```

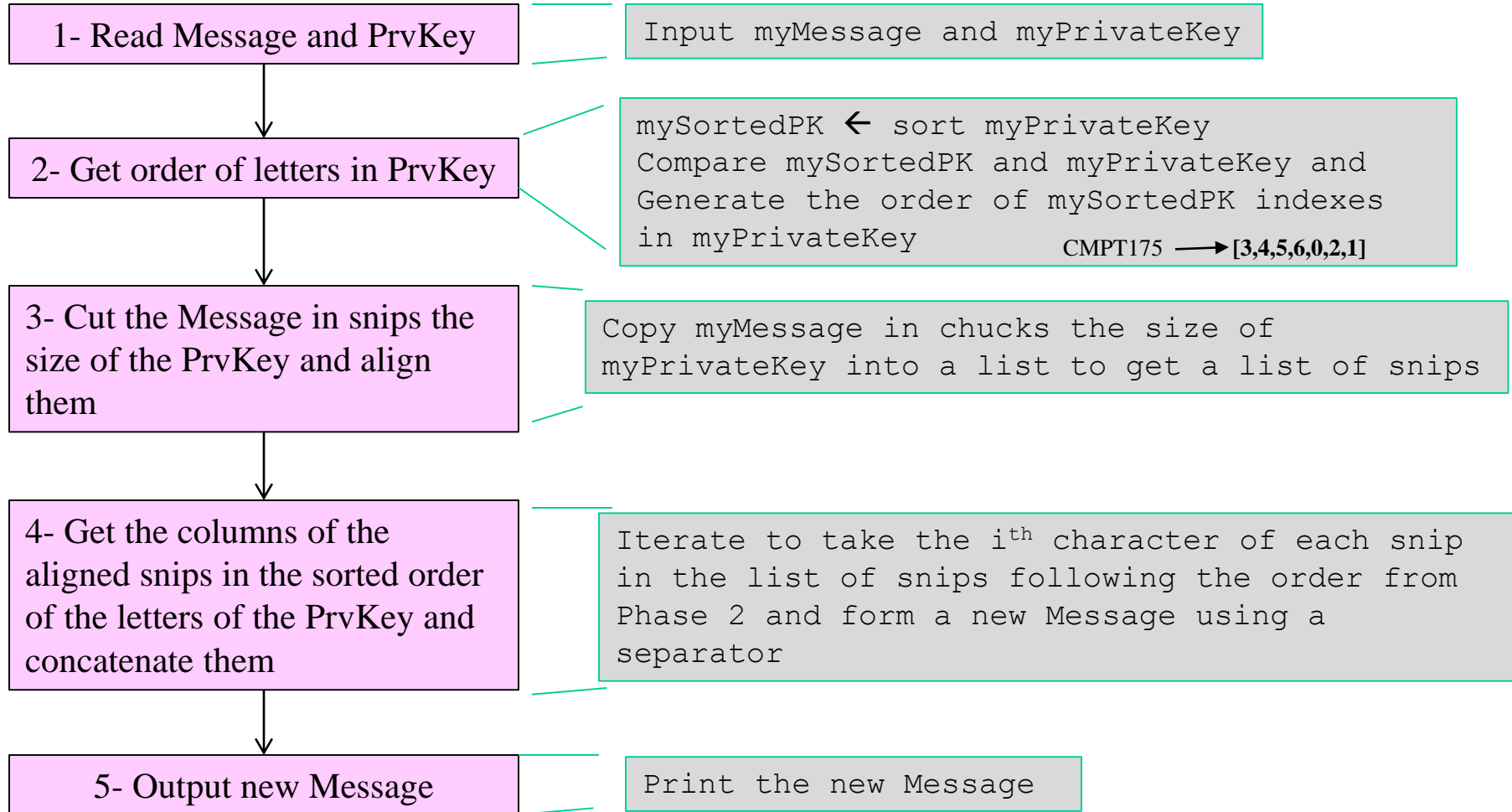
4-Get the columns of the aligned snips in the sorted order of the letters of the PrvKey and concatenate them

M						t
h						c
e						a
	y		e	r		
	e		w			
	i		t			
m	n	r	r			
f	p	v		e		
	o	e		H		
a	b	u		a		
h	y	r	.			
s	a	i				
m	o	.				
	e	s				
		s				
		s				

## 5-Output new Message

Cipher = “rrd ehtHa#tcao sd ea#oe crihewg#Mhe mf ahsm#yeinpobyaoe# rveuri.sms#b ewt r e.#”

# Simple Enciphering Algorithm





# Simple Enciphering in Python

Input myMessage & myPrivateKey

mySortedPK ← sort myPrivateKey  
Compare mySortedPK and myPrivateKey and Generate the order of mySortedPK indexes in myPrivateKey

Copy myMessage in chunks the size of myPrivateKey into a list to get a list of snips

Iterate to take the  $i^{\text{th}}$  character of each snip in the list of snips following the order from Phase 2 and form a new Message

Print the new Message

```
myMessage=input("Enter your Message:")  
myPrivateKey=input("Enter your Private Key:")
```

```
mySortedPK ← sort myPrivateKey  
For each character in myPrivateKey  
    check at what position it is in mySortedPK  
    and add that position to the order list
```

CMPT175 → [3,4,5,6,0,2,1]

```
mySortedPK= sorted(myPrivateKey) #returns a list  
myListedPK= list(myPrivateKey)   #creates a list  
myOrder=[]  
i=0  
while i<len(myListedPK):  
    j=mySortedPK.index(myListedPK[i])  
    myOrder.append(j)  
    mySortedPK[j]= ' '  
    i=i+1
```

Why adding a space here?

Duplicate characters in key

# Simple Enciphering in Python

myPrivateKey = "CMPUT175"

```
mySortedPK= sorted(myPrivateKey)
myListedPK= list(myPrivateKey)
myOrder=[]
i=0
while i<len(myListedPK):
    j=mySortedPK.index(myListedPK[i])
    myOrder.append(j)
    mySortedPK[j]= ' '
    i=i+1
```

mySortedPK  $\leftarrow$  [1,5,7,C,M,P,T]

myListedPK  $\leftarrow$  [C,M,P,T,1,7,5]

while  $i < 7$

When  $i$  is 0  $j \leftarrow$  index of 'C' in mySortedPK which is 3

Then when  $i$  is 1  $j \leftarrow$  index of 'M' in mySortedPK which is 4

Then when  $i$  is 2  $j \leftarrow$  index of 'P' in mySortedPK which is 5

And so on until

When  $i$  is 6  $j \leftarrow$  index of '5' in mySortedPK which is 1

At the end we get

myOrder is [3,4,5,6,0,2,1]

# Simple Enciphering in Python

```
myMessage = "My brother received a new computer for his birthday. He has awesome games."
```

Input myMessage & myPrivateKey

mySortedPK ← sort myPrivateKey  
Compare mySortedPK and  
myPrivateKey and Generate the  
order of mySortedPK indexes in  
myPrivateKey

Copy myMessage in chunks the  
size of myPrivateKey into a  
list to get a list of snips

Iterate to take the  $i^{\text{th}}$   
character of each snip in the  
list of snips following the  
order from Phase 2 and form a  
new Message using a separator

Print the new Message

```
i=0  
snips=[]  
while i<len(myMessage):  
    snip=myMessage[i:i+len(myPrivateKey)]  
    snips.append(snip)  
    i=i+len(myPrivateKey)
```

snips

```
["My brot",  
"her rec",  
"eived a",  
" new co",  
"mputer ",  
"for his",  
" birthd",  
"ay. He ",  
"has awe",  
"some ga",  
"mes."]
```

snip

```
i=0  My brot  
i=7  her rec  
i=14 eived a  
i=21 new co  
i=28 mputer  
i=35 for his  
i=42 birthd  
i=49 ay. He
```

...

```
i=70 mes.
```

Cipher = "rrd ehtHa#tcao sd ea#oe crihewg#Mhe mf ahsm#yeinpobyaoe# rveuri.sms#b ewt r e.#"

# Simple Enciphering in Python

```
Snips=["My brot","her rec","eived a"," new co","mputer ","for his"," birthd","ay. He ","has awe","some ga","mes."]
```

Input myMessage & myPrivateKey

mySortedPK ← sort myPrivateKey  
Compare mySortedPK and  
myPrivateKey and Generate the  
order of mySortedPK indexes in  
myPrivateKey

Copy myMessage in chunks the  
size of myPrivateKey into a  
list to get a list of snips

Iterate to take the  $i^{\text{th}}$   
character of each snip in the  
list of snips following the  
order from Phase 2 and form a  
new Message using a separator

Print the new Message

```
myOrder= [3,4,5,6,0, 2,1]
```

```
j=0; i=4
```

```
k=0; snip="My brot"; myEncodeMessage=['r']
```

```
k=1; snip="her rec"; myEncodeMessage=['r','r']
```

```
k=2; snip="eived a"; myEncodeMessage=['r','r','d']
```

```
k=3; snip=" new co"; myEncodeMessage=['r','r','d',' ']
```

```
myEncodedMessage=[]          # this is a list
j=0
while j<len(myOrder):        # for all the columns
    c=0
    while c<len(myOrder):    # traverse myOrder
        if j==myOrder[c]:    # find where j is
            i=c
        c=c+1
    k=0
    while k<len(snips):       # traverse one column
        snip=snips[k]
        myEncodedMessage.append(snip[i])
        k=k+1
    myEncodedMessage.append('#') # end of snip
    j=j+1
cipher=''.join(myEncodedMessage)
```

# Simple Enciphering in Python

Input myMessage & myPrivateKey

mySortedPK  $\leftarrow$  sort myPrivateKey  
Compare mySortedPK and myPrivateKey and Generate the order of mySortedPK indexes in myPrivateKey

Copy myMessage in chunks the size of myPrivateKey into a list to get a list of snips

Iterate to take the  $i^{\text{th}}$  character of each snip in the list of snips following the order from Phase 2 and form a new Message using a separator

Print the new Message

print (cipher)

```
i=0
snips=[]
while i<len(myMessage):
    snip=myMessage[i:i+len(myPrivateKey)]
    i=i+len(myPrivateKey)
    snips.append(snip)
```

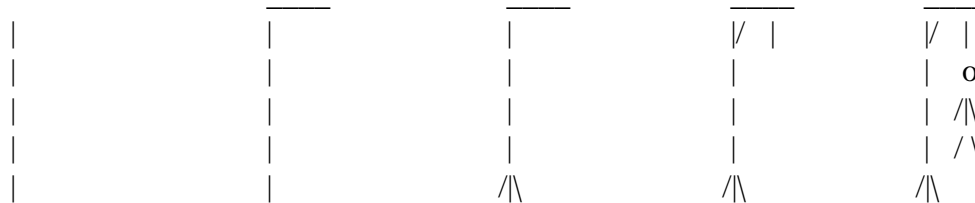
```
myEncodedMessage=[] # this is a list
j=0
while j<len(myOrder): # for all the columns
    c=0
    while c<len(myOrder): # traverse myOrder
        if j==myOrder[c]: # find where j is
            i=c
            c=c+1
        k=0
        while k<len(snips): # traverse one column
            snip=snips[k]
            myEncodedMessage.append(snip[i])
            k=k+1
        myEncodedMessage.append('#') # end of snip
        j=j+1
cipher=''.join(myEncodedMessage)
```

# Other details in the program

- What if the original message contains “#”?
- Should we only use one separator #?
- Could we alternate between separators?
- What about a long message in a file?
- Deciphering is the exact reverse procedure
- More details in the complete python code on the course eclass page

# Hangman Word Game

- Hangman is a word guessing game. One player thinks of a word, and the other tries to guess it by suggesting letters.
- The word to guess is represented by a row of dashes, giving the number of letters: \_ \_ \_ \_
- When a letter is guessed correctly it is written in its position; when a letter is incorrectly suggested a section of the gallows is drawn.
- The game ends when the player guesses and completes the word or the hang man is drawn on the gallows. The hangman and gallows are drawn in 5 steps (i.e. 5 wrong guesses)



# Input of the problem



- There is a text file of words
- Each line of the file contains a word
- E.g. Cat
- We will traverse the file sequentially
- Letter guesses are entered on the keyboard





# Expected Output

- Initially
  - Dashes indicating the number of letters in the word  
eg: \_ \_ \_ (assume *cat* is the word)
- During the guesses (assuming *a* was suggested)
  - Dashes where there still missing letters eg: \_a\_
  - Suggested letters in their correct position eg: \_a\_
  - List of wrong suggestions in the order they were suggested. eg: h, e, s, o
  - The drawing of the hangman at the step equal to the number of wrong suggestions.
- When the game is over
  - Display the word and either R.I.P. if hangman is finished or "Well done" if the word is correctly guessed.



## Examples of Output

Guess a letter

- In the beginning

```

| / | _ o _ _ u _ e r
|
|
|
| a, i, h, b
/ | \ Guess a letter

```

- During the game

The guess is incorrect

✓ The guess is correct

```

|/      |  _ o m _ u _ e r
|        o
|      /|\
|      / \   a, i, h, b, l
/|\      R.I.P.
          The word was "Computer"

```

```

|/      | C o m p u t e r
|
|
|      a, i, h, b
/|\     Computer.
        Well Done!

```

# Process

## Initialization

- Read the file to get a word
- Display dashes

## Game

- Repeat until game over
  - Get letter suggestion
  - If letter in word, display letter in position
  - If letter not in word, draw gallows

## Conclusion

- If word guessed, display "well done"
- If word not guessed, display "R.I.P."

# Visualizing the problem



Read  
word W

Display *Template*

repeat

When  
to  
stop?

- 1- Read character C
- 2- Correct → update Template
- 3- NOT Correct →
  - 3.1- Update Gallows
  - 3.2- Append C in List of wrong guesses
- 4- Display *Template*, *Gallows*, and List

Display RIP

Display Well Done

[example]

Template is a

List of wrong guesses

is h, e, s, o

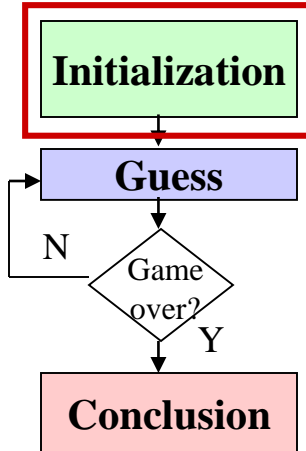
Gallows is



# Pseudo-code

- Initialization

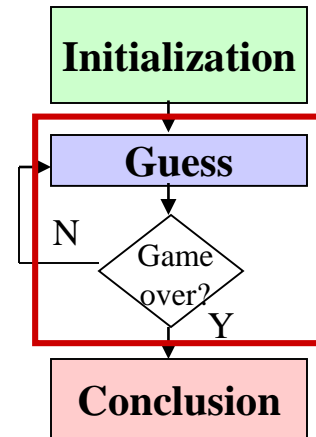
- Open file;
- Read one word  $W$  from file;
- Close file;
- Check length of  $W$
- Initialize template  $\leftarrow$  string ( $W.length$ , “\_”)
- Initialize hangman.step  $\leftarrow 0$
- Initialize game\_over  $\leftarrow$  false
- Initialize wrong\_guesses  $\leftarrow \{\}$ ;



# Pseudo-code

- Repeated Guess

- Display template;
- Display hangman;
- Display wrong\_guesses;
- Read Character C from keyboard
- If Exists(C, W)
  - Update(template,C)
- Else
  - $\text{hangman.step} \leftarrow \text{hangman.step} + 1;$
  - Append (C, wrong\_guesses)
- $\text{game\_over} \leftarrow (\text{hangman.step}=5) \text{ or } (\text{template.full})$



# Pseudo-code

- Conclusion

- If template.full

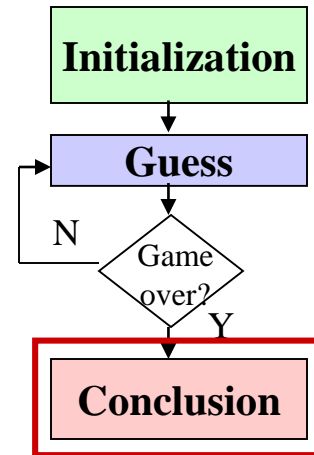
- Display hangman;
    - Display W

- Display "Well Done"

- else

- Display hangman;
    - Display W

- Display "R.I.P."



- Display hangman;
- Display W
- If template.full
  - Display "Well Done"
- else
  - Display "R.I.P."

# Data Structure Needed

• W	String
• C	Character
• template	Object
• hangman	Object
• game_over	Boolean
• wrong_guesses	List



# Objects

Instance from Class Template

## 🍌 template

Displays the template

show()

Changes the template  
position with  
character if in word

update(word,character)  
Returns True/False

Checks whether all  
letters are guessed

isFull()  
Returns True/False

dashes

List

## 🍌 hangman

Displays the gallows

show()

Inquires about the  
number of wrong  
guesses

get()  
Returns Integer

Adds 1 to step

increment()

step

Integer

Instance from Class Gallows

# Implementation of classes with Python

```
### class Template
```

```
class Template:
```

```
    def __init__(self, size):
```

```
        self.dashes = ["_"]*size
```

```
    def isFull(self):
```

```
        if "_" in self.dashes: return False
```


```
        else: return True
```

# Implementation of classes with Python

```
def update(self, word, character):  
    found=False  
    for i in range(len(word)):  
        if character == word[i]:  
            self.dashes[i]=character  
            found=True  
    return found
```

Create a template then  
show it.  
Update the template then  
show it.  
Check if it is full, etc.

Remember to test the  
class and each method  
independently from the  
program.

```
def show(self):  
    line = ""   
    for i in range(len(self.dashes)):  
        line += self.dashes[i] + "  
    print (line)
```

# Steps for the Gallows

	Step 1	Step 2	Step 3	Step 4	Step 5
L1		_____	_____	_____	_____
L2				/	/
L3					○
L4					/   \
L5					/ \
L6			/   \	/   \	/   \

- **L2** to **L6** are always at least “|”
- Except for **Step 1** **L1** is always “\_\_\_\_\_”
- Above **Step2** **L6** has a base “/|”
- Above **Step3** **L2** has reinforcement and rope “/ | ”
- In **Step5** the man is hung

# Implementation of classes with Python

```
### class Gallows
```

```
class Gallows:
```

```
    def __init__(self):  
        self.step = 0
```

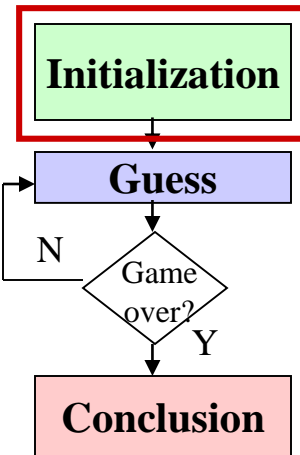
```
    def increment(self):  
        self.step += 1
```

```
    def get(self):  
        return self.step
```

```
    def show(self):  
        l1=l2=l3=l4=l5=l6=""  
        if self.step>0:  
            l2=l3=l4=l5=l6=" |"  
        if self.step>1 :  
            l1=" ____"  
        if self.step>2:  
            l6="/|\\"  
        if self.step>3:  
            l2=" |/"  
        if self.step==5:  
            l3=" | o"  
            l4=" | /|\\"  
            l5=" | /|\\"  
        print (l1,l2,l3,l4,l5,l6,sep="\n")
```

# Implementation with Python

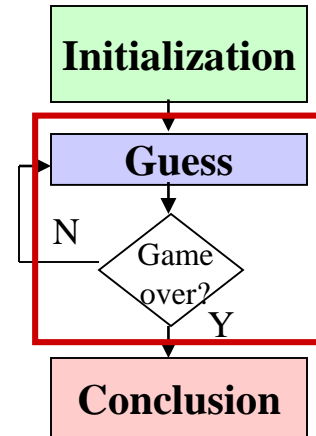
```
### hangman word game
myfile = open('wordfile', 'r')
myWord = myfile.readline()
myfile.close()
wrongGuesses = []
gameOver = False
myWord=myWord.rstrip()
myTemplate = Template(len(myWord))
hangman = Gallows()
```



Notice only  
1<sup>st</sup> word is  
read. A better  
solution is  
needed

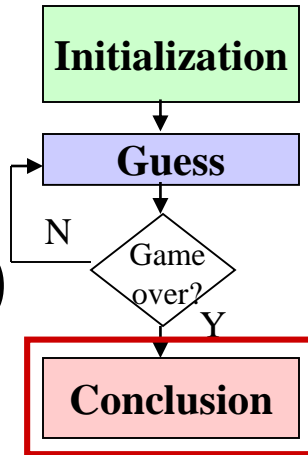
# Implementation with Python

```
while not gameOver:
    hangman.show()
    myTemplate.show()
    out = ""
    for i in range(len(wrongGuesses)):
        out += wrongGuesses[i] + ", "
    print(out)
    theInput=input("Guess a letter: ")
    myChar=theInput[0]
    if not myTemplate.update(myWord,myChar):
        hangman.increment()
        wrongGuesses.append(myChar)
    gameOver = myTemplate.isFull() or hangman.get() == 5
```



# Implementation with Python

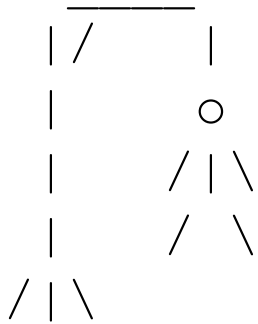
```
if myTemplate.isFull():  
    print ("Well Done!")  
    print ("The Word was indeed " + myWord)  
else:  
    hangman.show()  
    myTemplate.show()  
    out = ""  
    for i in range(len(wrongGuesses)):  
        out += wrongGuesses[i] + ", "  
    print (out + "R.I.P.!")  
    print ("The Word was " + myWord)
```





# Testing

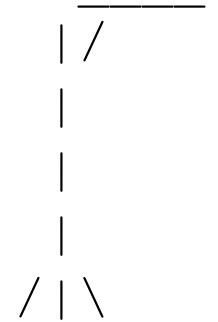
-----  
Guess a letter:



c \_ \_ \_ \_ t e r

x, g, h, y, k,  
R.I.P.!

The Word was computer



c \_ \_ \_ \_ t e r

x, g, h, y,  
Guess a letter:

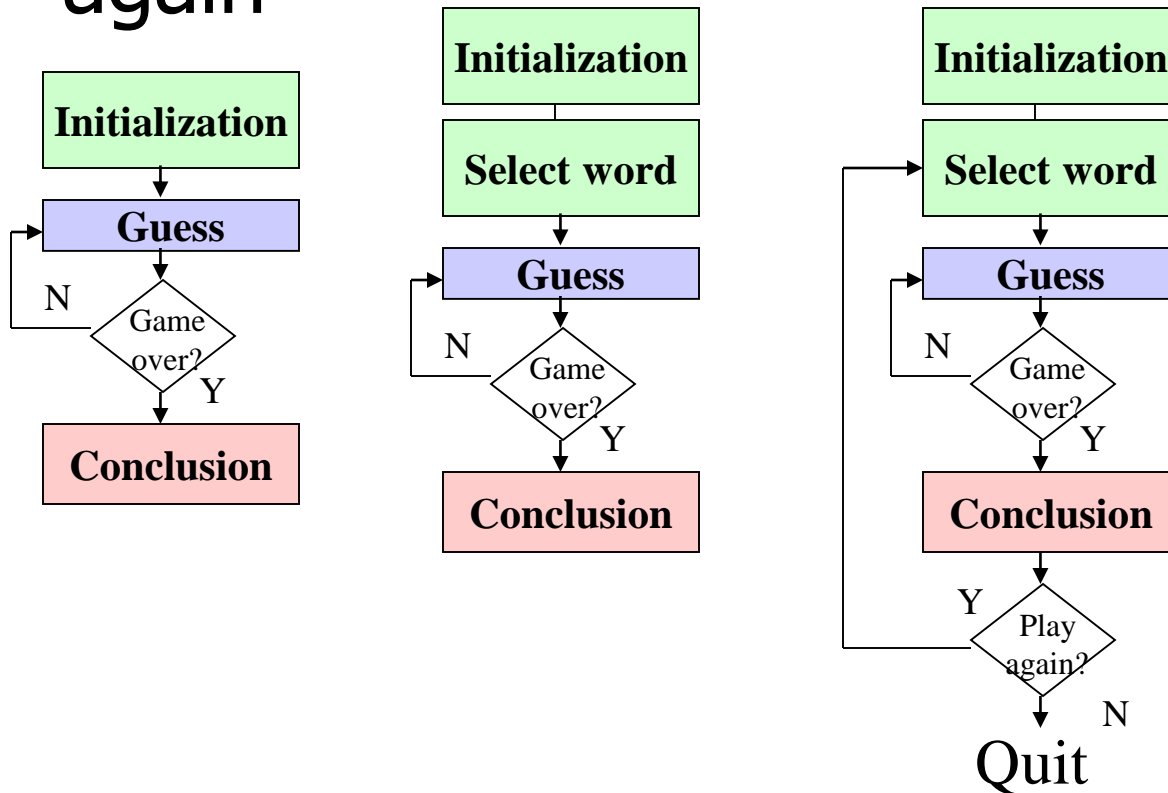
Well Done!

The word is indeed: computer

# Repeating the game



- Ask whether the player wants to play again



# Adding a Global Loop

```
### hangman word game
myfile = open('wordfile', 'r')
playingGame = True
while playingGame:
    myWord = myfile.readline()
    wrongGuesses = []
    gameOver = False
    myWord=myWord.rstrip()
    myTemplate = Template(len(myWord))
    hangman = Gallows()

...

```

# End of Global

...

```
while True:
```

```
    response=input("Do you want to play again? (Y/N)")
```

```
    play=response[0]
```

```
    if play.upper()=="Y" or play.upper()=="N": break
```

```
    if play.upper()=="N": playingGame=False
```

```
myfile.close()
```

```
print("Thank you for playing Hangman")
```

# Improvements



1. Each line of the file contains a word and a hint. These are comma separated
  - E.g. Cat, animal
  - Display the hint with the template
2. Select randomly within the file
3. The list of wrong guesses should be sorted alphabetically
4. A wrong guess should not be entered again
5. Only valid input should be accepted.

horse,animal  
orange,fruit  
tree,living thing  
bee,animal  
keyboard,object  
computer,object  
machine,object  
automobile,object  
canada,country  
...

# Splitting the input line

```
myfile = open('wordshint.txt', 'r')
myLines=myfile.readlines()
myfile.close()

...
i=random.randrange(0,len(myLines))
myLine = myLines[i].rstrip()
[myWord,myHint]=myLine.split(",",1)

...
wrongGuesses.append(myChar)
wrongGuesses=sorted(wrongGuesses)
```

```
import random
```

```
class Gallows:
```

```
# Implements the gallows for the hangman
```

```
def __init__(self):
    self.step = 0
```

```
def increment(self):
    self.step += 1
```

```
def get(self):
    return self.step
```

```
def show(self):
    # print the gallows depending on step
    l1=l2=l3=l4=l5=l6=""
    if self.step>0:
        l2=l3=l4=l5=l6=" |"
    if self.step>1 :
        l1="  _  "
    if self.step>2:
        l6="/\\\\"
    if self.step>3:
        l2=" / | "
    if self.step==5:
        l3=" | o"
        l4=" | /\\\\"
        l5=" | /\\\\"
    print (l1,l2,l3,l4,l5,l6,sep="\n")
```

```
class Template:
```

```
# Implements the template to fill with the word to guess
```

```
def __init__(self,size):
    self.dashes = ["_"]*size
```

```
def isFull(self):
    # it is full when all dashes are eliminated
    if "_" in self.dashes: return False
    else: return True
```

```
def update(self, word, character):
    # place character in template if in word
    # returns True if character in word
    found=False
    for i in range(len(word)):
        if character == word[i]:
            self.dashes[i]=character
            found=True
    return found
```

```
def existIn(self,char):
    # returns True if char is already in template
    if char in self.dashes: return True
    else: return False
```

```
def show(self):
    # prints the current state of the template
    line = ""
    for i in range(len(self.dashes)):
        line += self.dashes[i] + " "
    print (line)
```

# Python code for simple hangman game 2/2

# Opening file and reading it in memory

```
myfile = open('wordshint.txt', 'r')
myLines=myfile.readlines()
myfile.close()
```

playingGame=True

# Main loop of the game

while playingGame:

    wrongGuesses = []

    gameOver = False

    # Selecting a random word

    i=random.randrange(0,len(myLines))

    myLine = myLines[i].rstrip()

    [myWord,myHint]=myLine.split(",",1)

    myTemplate = Template(len(myWord))

    hangman = Gallows()

```
while not gameOver :
    hangman.show()
    myTemplate.show()
    print ("Hint:",myHint)
    out=""
    for i in range(len(wrongGuesses)):
        out += wrongGuesses[i] + ", "
    print (out)
    accept=False # accept a character not entered before. Repeat until alpha and new
    while not accept:
        theInput=input("Guess a letter: ")
        myChar=theInput[0]
        if myChar not in wrongGuesses and not myTemplate.existIn(myChar) and
myChar.isalpha(): accept = True
        if not myTemplate.update(myWord,myChar):
            hangman.increment()
            wrongGuesses.append(myChar)
            wrongGuesses=sorted(wrongGuesses)
        gameOver = myTemplate.isFull() or hangman.get()>=5

# time is up. If the template is filled then the player won, otherwise dead
if myTemplate.isFull():
    print ("Well Done! The word is indeed: ", myWord)
else:
    hangman.show()
    myTemplate.show()
    print ("R.I.P! The Word was [", myWord, "]")
while True: # get either a Y or N whether to continue the game
    response=input("Do you want to play again? (Y/N)")
    play=response[0]
    if play.upper()=="Y" or play.upper()=="N": break
    if play.upper()=="N": playingGame=False
print("Thank you for playing Hangman")
```



# More suggestions for output

Line 1  
 Line 2  
 Line 3        \_ o \_ \_ u \_ e r  
 Line 4  
 Line 5        a, i, h, b  
 Line 6        /|\        Guess a letter

Line 1  
 Line 2  
 Line 3        o        \_ o m \_ u \_ e r  
 Line 4        /|\  
 Line 5        / \        a, i, h, b, l  
 Line 6        /|\        R.I.P.  
 Line 7               The word was "Computer"

Line 1  
 Line 2  
 Line 3  
 Line 4  
 Line 5        \_ o \_ \_ u \_ e r  
 Line 6        a, i, h, b,  
 Line 7        Guess a letter:

**As opposed to**