Abstraction

* Simplifies things
* Identifies what is important
* Manages Complexity

Program = Algorithm + Data Structures

Abstractions in Algorithms – Functions

Abstractions in Data – Data Structures (Strings, Lists, ect)

**Python Strings**

A string is a sequence of characters.

* Defined by ‘ ’ and “ “

Apostrophe in strings:

* “Mike’s book
* ‘Mike\’s book’

H E L L O W O R L D

0 1 2 3 4 5 6 7 8 9 10

… -3 -2 -1

mystr[1] = E

mystr[1:4] = ELL (remember that the last number is not included)

mystr[-7:-1] = O WORL

mystr[-3:-5] = ‘’ (can’t go backwards. Empty.)

mystr[5:] = ‘ WORLD’

mystr[3:-3] = LO WO (combined numbering systems)

mystr[::2] = HLOWRD (step)

Commands to Copy and Reverse strings

astr = “string to copy”

newstr = astr[:] (This obtains a list of characters)

newstr = ‘’.join(astr) (This obtains a list of characters joined by ‘’, or nothing)

revstr = astr[::-1] 🡪 ypoc ot gnirts (this reverses the string)

*note that python auto-reverses the start/end*

failrev = astr[0:14:-1] 🡪 ‘’ (this fails because the start/end are not reversed)

Basic Operations

str = “Basic”

len(str) = 5

str + “operations” = “Basic operations”

str \* 3 = “BasicBasicBasic”

Code to represent characters

ASCII

* 8 bit (256 characters)
* limited

Unicode

* 16 bit (65636 characters)
* 17 planes of characters
* Total of 1 114 112 characters
* Only 96 000 used.

ASCII

Lowercase > Uppercase > Numbers

* Lowercase has a higher ASCII code

ord(‘a’) = 97 (same for standard ASCII and Unicode symbols)

chr(97) = ‘a’

One use is encryption.

code = ord(‘a’)

chr(code + 1) (a becomes b)

‘a’ == ‘a’ 🡪 True

‘a’ < ‘b’ 🡪 True

‘1’ < ‘9’ 🡪 True

‘a’ < ‘B’ 🡪 False

* Basically, each character has a numerical value
* These can be added together.

Operator: in

str = abcdefg

‘c’ in str 🡪 True

‘cef’ in str 🡪 False

str in str 🡪 True

Strings are immutable. You cannot change strings.

str = ‘spam’

str[1] = l 🡪 Error

newstr = str[:1] + ‘l’ + str[2:]

newstr = ‘slam’

str = ‘spam

newstr = str.replace(‘a’, ’z’) (copy string, replace all ‘a’ with ‘z’)

*note:*

str = ‘slam’ 🡪 You have overriden the string

String Methods

**len()**

* Length includes spaces.

**.upper()**

str = ‘shouting!’

str.upper() 🡪 ‘SHOUTING!’

Object – str

Method – upper()

Method call – mystr.upper()

Therefore, the general case is: object.method()

**find()**

str = “Find in a string”

str.find(‘d’) 🡪 3

**join()**

str1 = “abcd”

str2 = “1234”

str2.join(str1) 🡪 “a1234b1234c1234d”

str1.join(str2) 🡪 “1abcd2abcd3abcd4”

base.join(target)

Basically, the base is placed between each item in (target)

**Composite Types (Data Structures)**

A new data type made from different ones.

Eg. Tuples, Lists, Dictionaries, Strings

* Convenient ways to store data

Lists

Create an empty list with:

list() (not iterable)

list(arg) (iterable)

eg. list(‘abc’) 🡪 [‘a’,’b’,’c’]

[1,2,3] + [4] 🡪 [1,2,3,4]

+ lists can be concatenated

\* lists can be repeated

[] lists can be indexed

[:] lists can be sliced

in lists can be checked

len() lists can be measured

Lists can hold numbers and Booleans, unlike strings

Lists are mutable, unlike strings

Indexing is the same as strings:

0 1 2 3

-4 -3 -2 -1

[0][0] index 0 of a list with one item (0)

* Indexes always come at the end of a variable or sequence.

List of Lists

myLst = [‘a’, [1,2,3], ‘a’]

myLst[1][0] 🡪 1

[1,[2,[3,4]],5][1][1][0] 🡪 3

List Functions

len [1,[1,2],3] 🡪 3

* Len() only takes the top level

min(lst)

max(lst)

* works on ‘a’ and ‘b’ due to Unicode

sum(lst)

* numeric only

Iterate on List

for elem in [1,[1,2], ‘a’, True]:

print(elem)

🡪 1

[1,2]

‘a’

True

Lists are mutable, so they can be changed.

mylist[0] = ‘a’

mylist.append(e) 🡪 Append ‘e’ to the back

mylist.extent(L) 🡪 Append list ‘L’ to the back

mylist.pop(4) 🡪 remove an index from the list (default is the last, -1)

mylist.insert(4,e) 🡪 insert e at index [4]

mylist.remove(e) 🡪 remove the first e

mylist.sort() 🡪 sort list

mylist.reverse() 🡪 reverse list

*note:*

lst = [6,5]

list.append(5) 🡪 [6,5,5]

list.extend(5) 🡪 ERROR. The ‘.extend’ expects a list.

list.append([8,9]) 🡪 [6,5,5, [8,9]] (A list is appended, whereas extend just adds elements)

*note:*

lst = [0,’b’]

lst.insert(0,1) 🡪 [1,0,’b’] (insert puts before that index)

lst.insert(-1,’a’) 🡪 [1,0,’a’,’b’] (BEFORE the index. -1 does NOT mean element is now at -1)

lst.insert(10,’c’) 🡪 [1,0,’a’,’b’,’c’] (setting a high number is fine.)

*note:*

You cannot sort a list with multiple types 1 and ‘a’.

Return Values

Some commands modify a list in place.

This means they do not **return** anything.

* Lists are mutable so there is no need for a new list.

lst = [4,7,1,2]

lst.sort() (sorts the list)

lst 🡪 [1,2,4,7]

lst = lst.sort() (assigns the return of .sort() to lst)

lst 🡪 None (this is bad)

Split

“This is a test”.split()

🡪 [“This”, “is”, “a”, “test”]

* Split returns a list.

Default is white space.

Only lists have a sorting method.

If you need to sort anything, convert it to a list.

string = ‘xyzabc’ (this is a string, immutable)

lst = list(‘xyzabc’)

lst.sort()

string = ‘’.join(lst) (this is a string again)

Tuple

Immutable list.

* Provides permanency.
* Avoid accidents.

Things that work on tuple:

* Indexing
* Slicing
* Len()
* Print()

The parenthesis is just the grouping.

The comma creates the tuple.

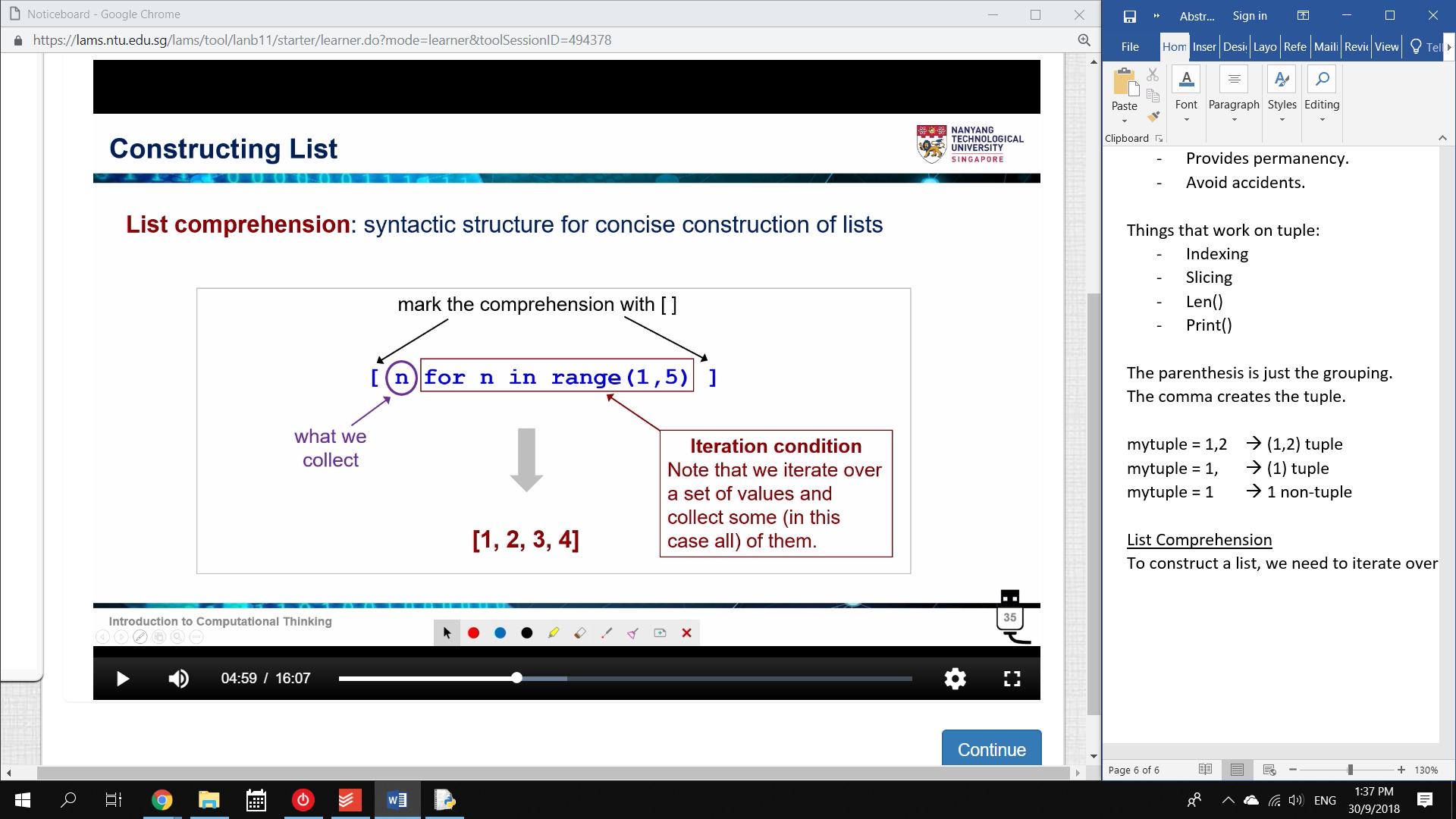
mytuple = 1,2 🡪 (1,2) tuple

mytuple = 1, 🡪 (1) tuple

mytuple = 1 🡪 1 non-tuple

List Comprehension

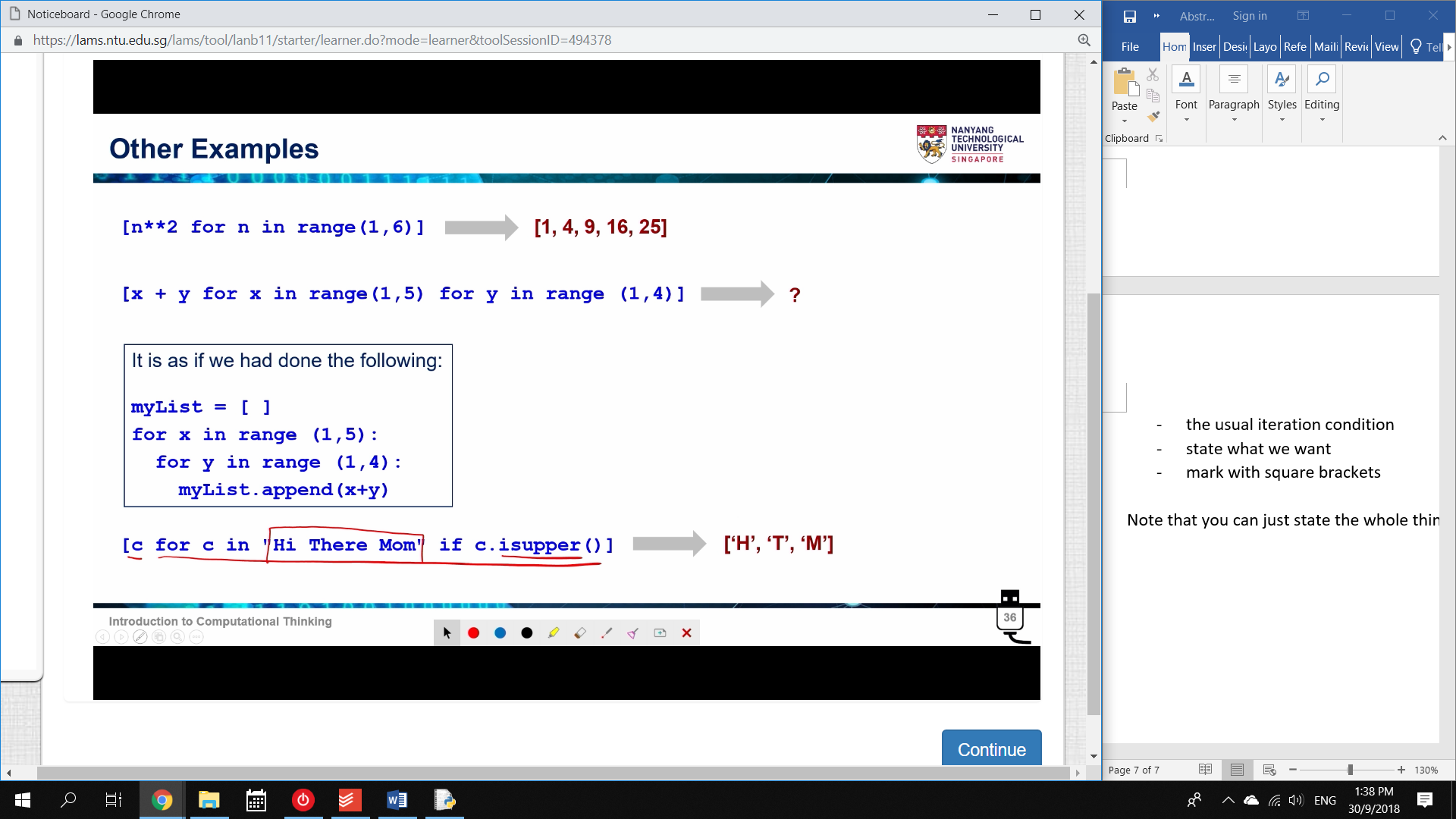
To construct a list, we need to iterate over a set of values.



n for n in range (1,5)

* the usual iteration condition
* state what we want
* mark with square brackets

Note that you can just state the whole thing in square brackets.



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

when y is 1, when x is 1, = 2

when y is 2, when x is 1, = 3

…

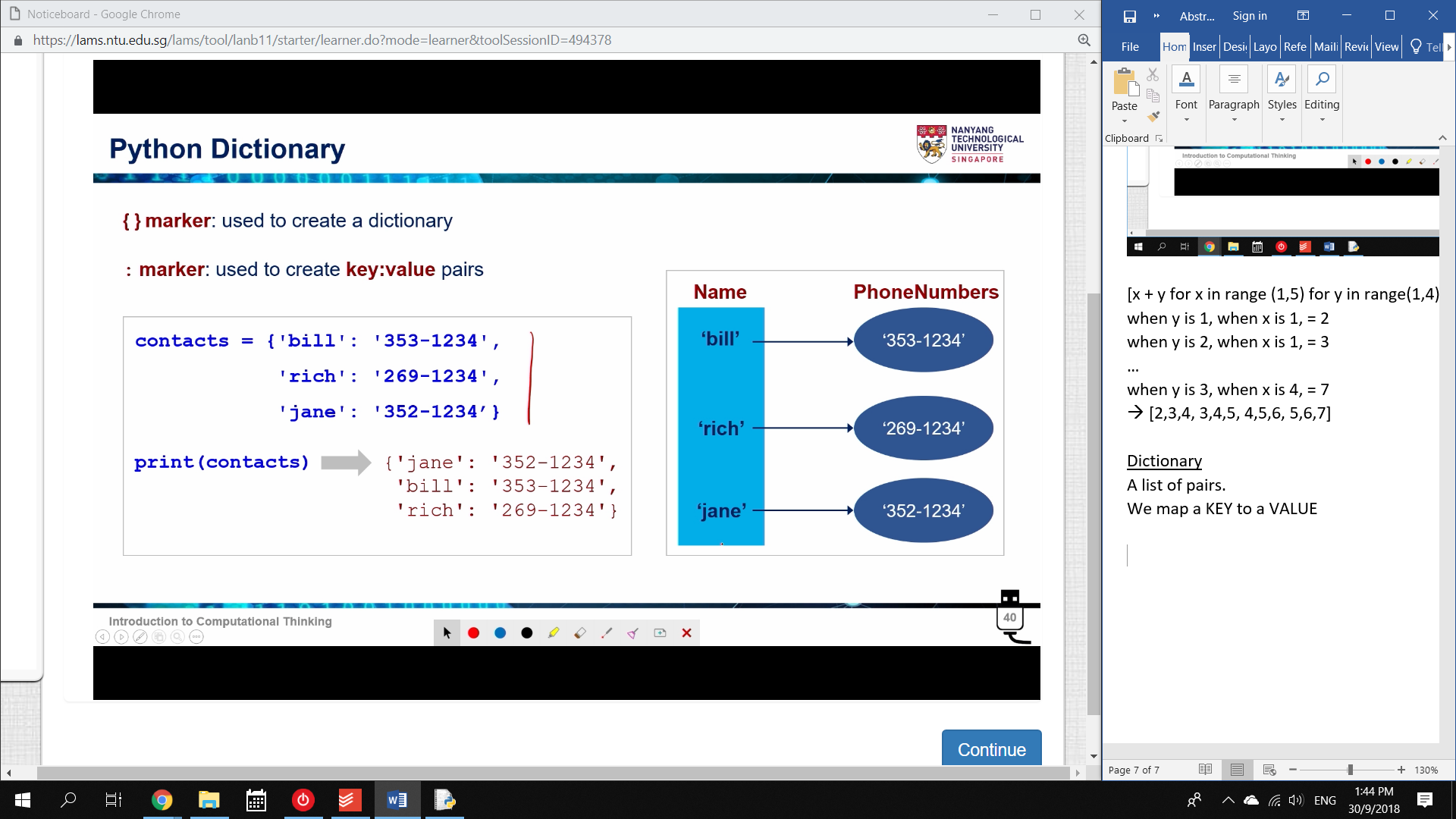
when y is 3, when x is 4, = 7

🡪 [2,3,4, 3,4,5, 4,5,6, 5,6,7]

Dictionary

A list of pairs.

We map a KEY to a VALUE



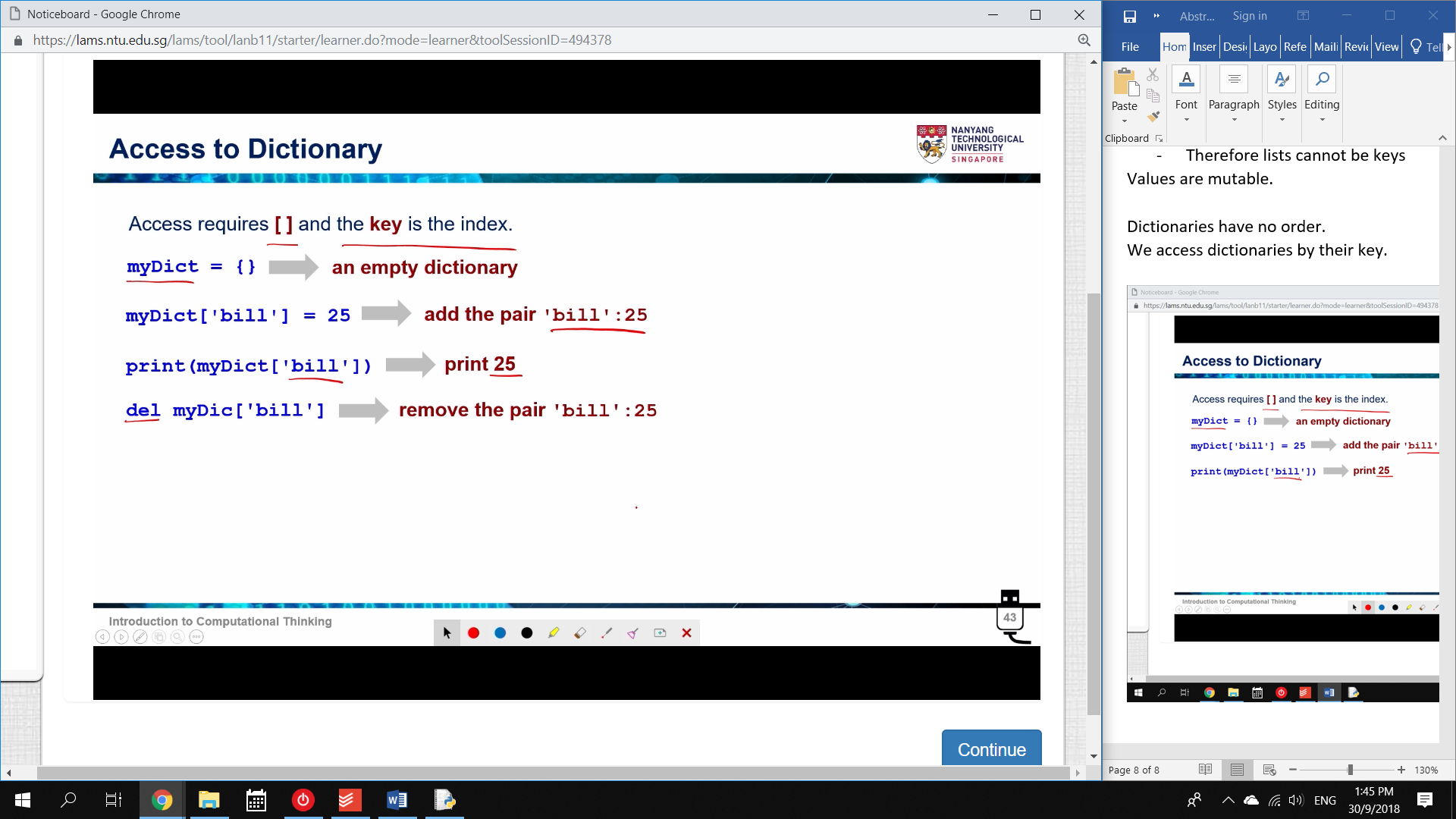
Keys are immutable.

* Therefore lists cannot be keys

Values are mutable.

Dictionaries have no order.

We access dictionaries by their key.

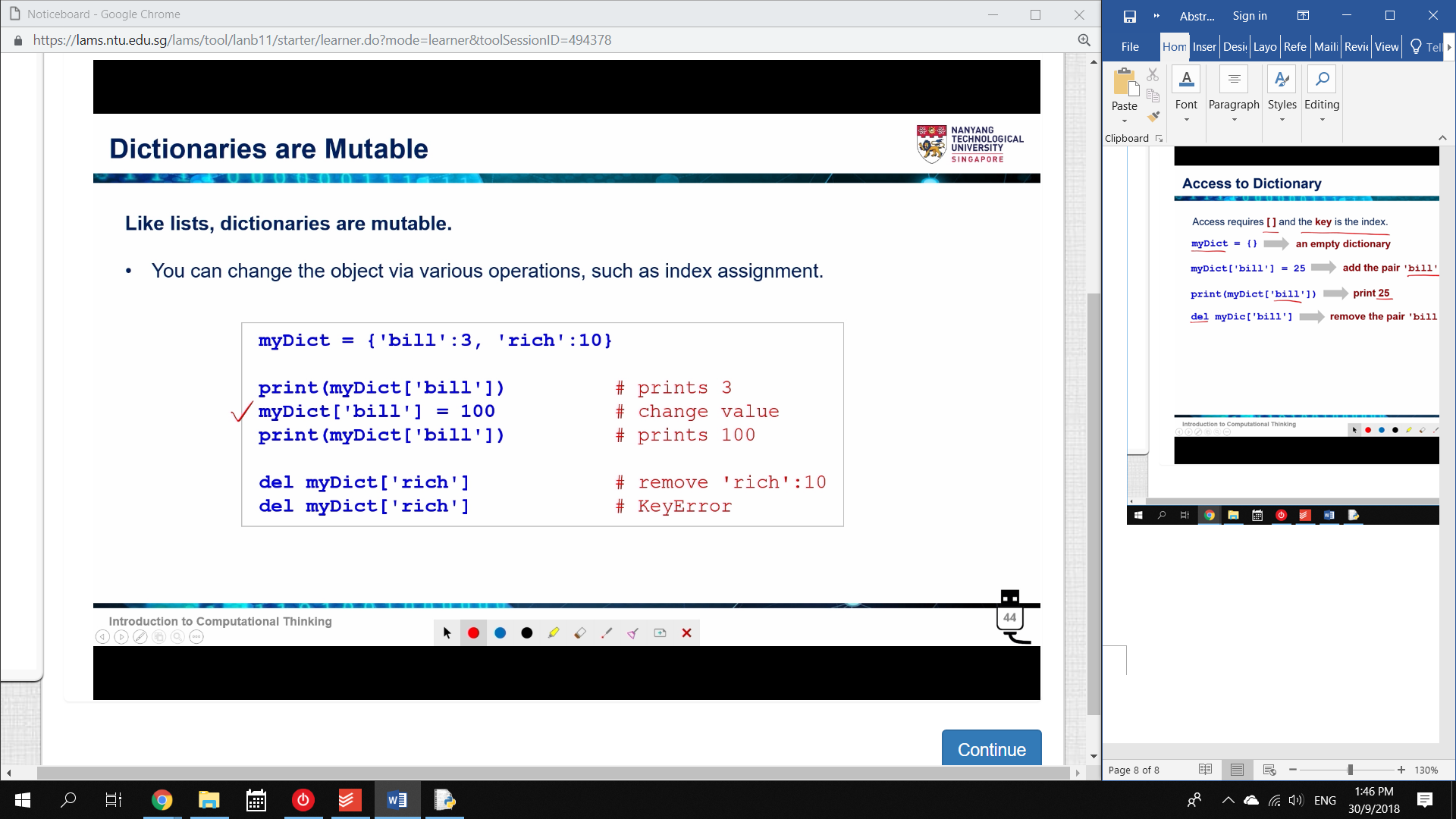


Accessing a dictionary is exactly the same as a list, except that the index is a Key.

my\_dict[‘bill’] = 25 (creates a key-value pair)

my\_dict[‘bill’] 🡪 25 (can be retrieved)

* basically, {bill:25} has been added.

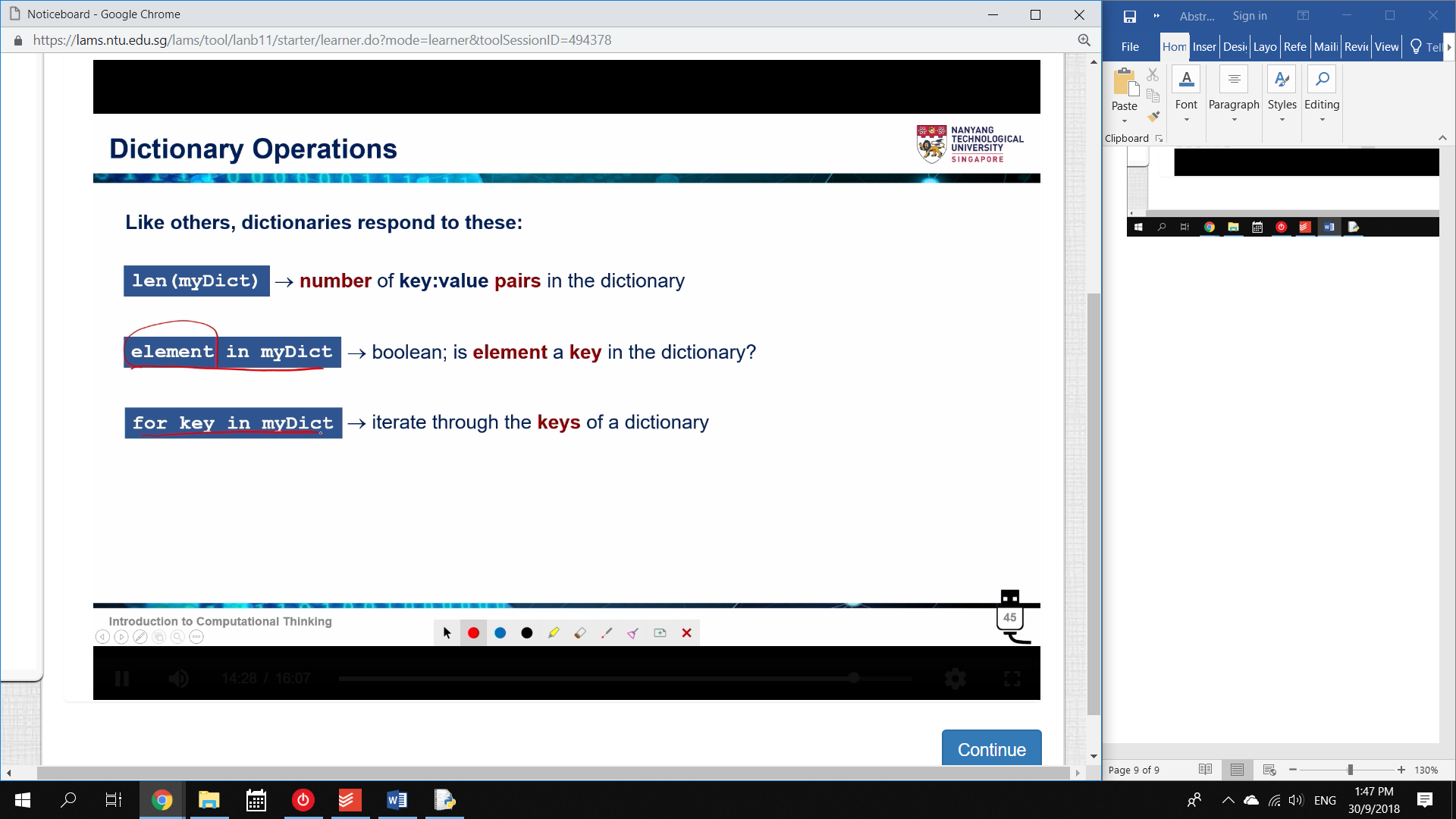


The value of my\_dict[bill] can be overridden at any time.

* just assign a new value

A key-value pair can be deleted at any time.

* deleting my\_dict[‘rich] deletes the whole pair.

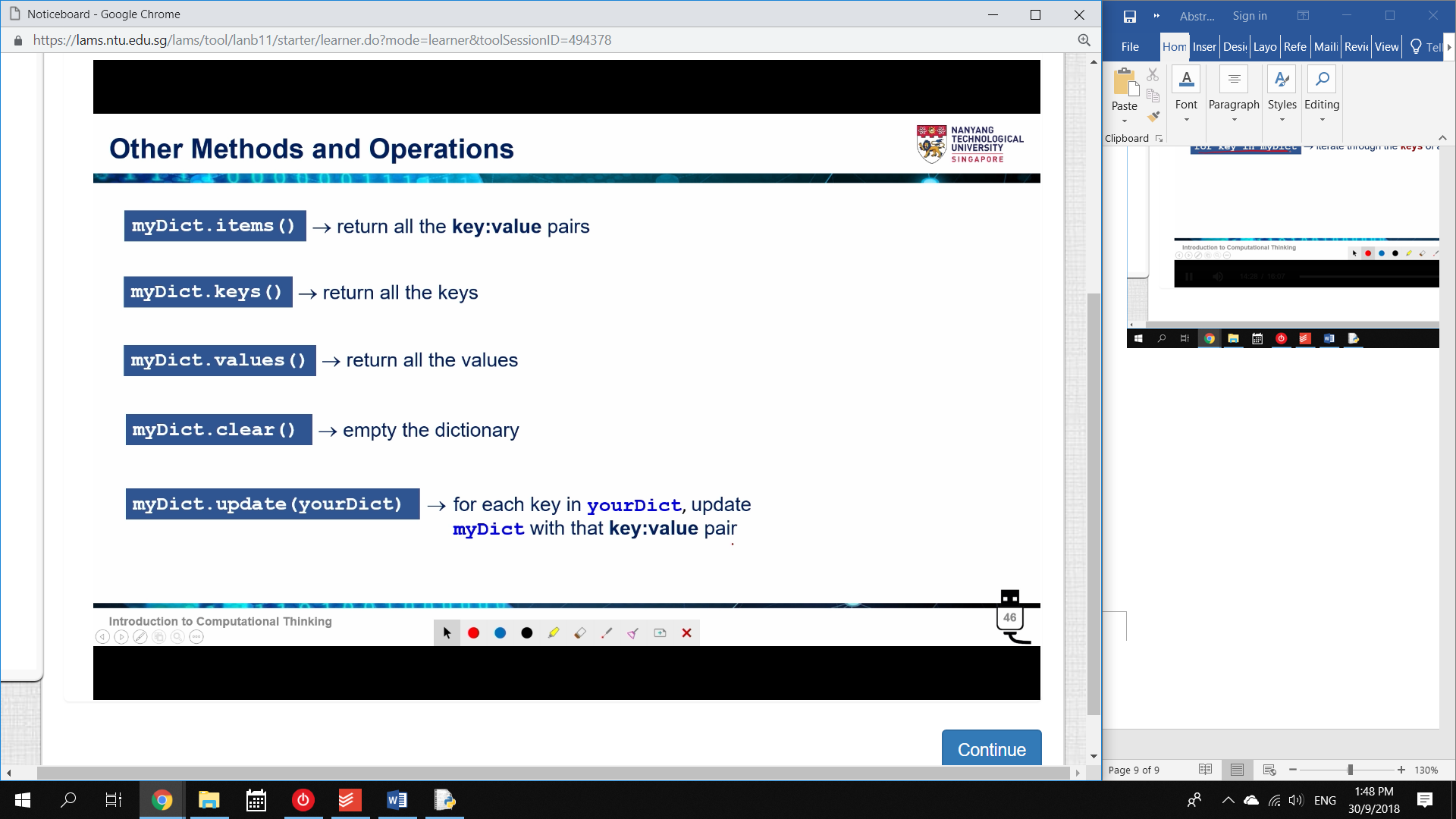


len() measures the number of pairs.

‘bill’ in my\_dict 🡪 True

25 in my\_dict 🡪 False

* Therefore, checking a dictionary checks the KEY not the VALUE



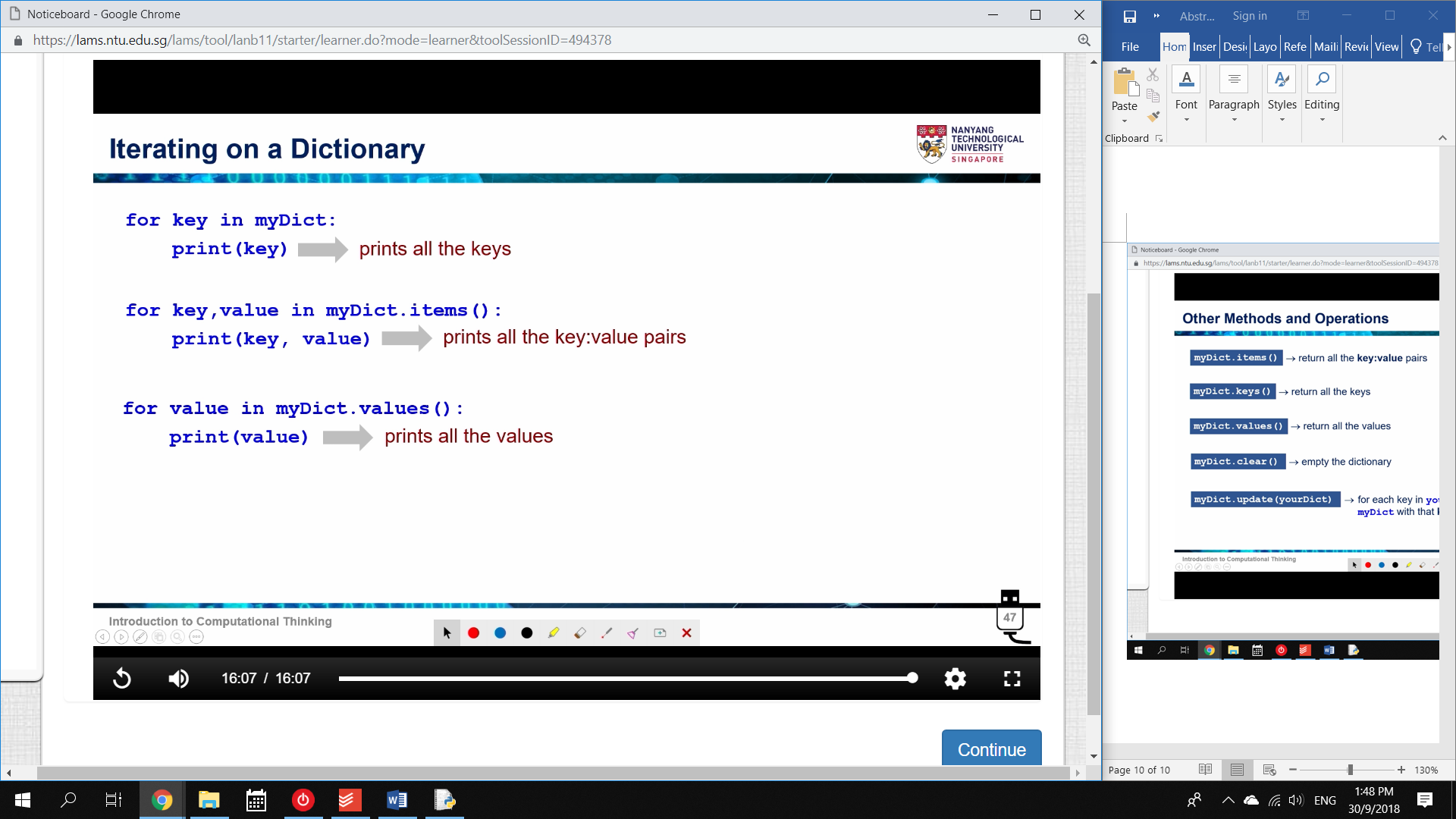
To check through values, we enter:

25 in my\_dict.values() 🡪 True

items() will iterate through all pairs.

clear() empties the dictionary

update(dict2) 🡪 all of dict2 contents will be added.



The second one works by the multiple assignment method.

* .items() is a pair.

**Function Development**

A function performs an operation and returns a value.

They allow:

* Abstraction
* Divide-and-Conquer problem solving
* Reuse
* Sharing
* Security
* Simplify programs and increase readability

def celsius2Farenheit(C)

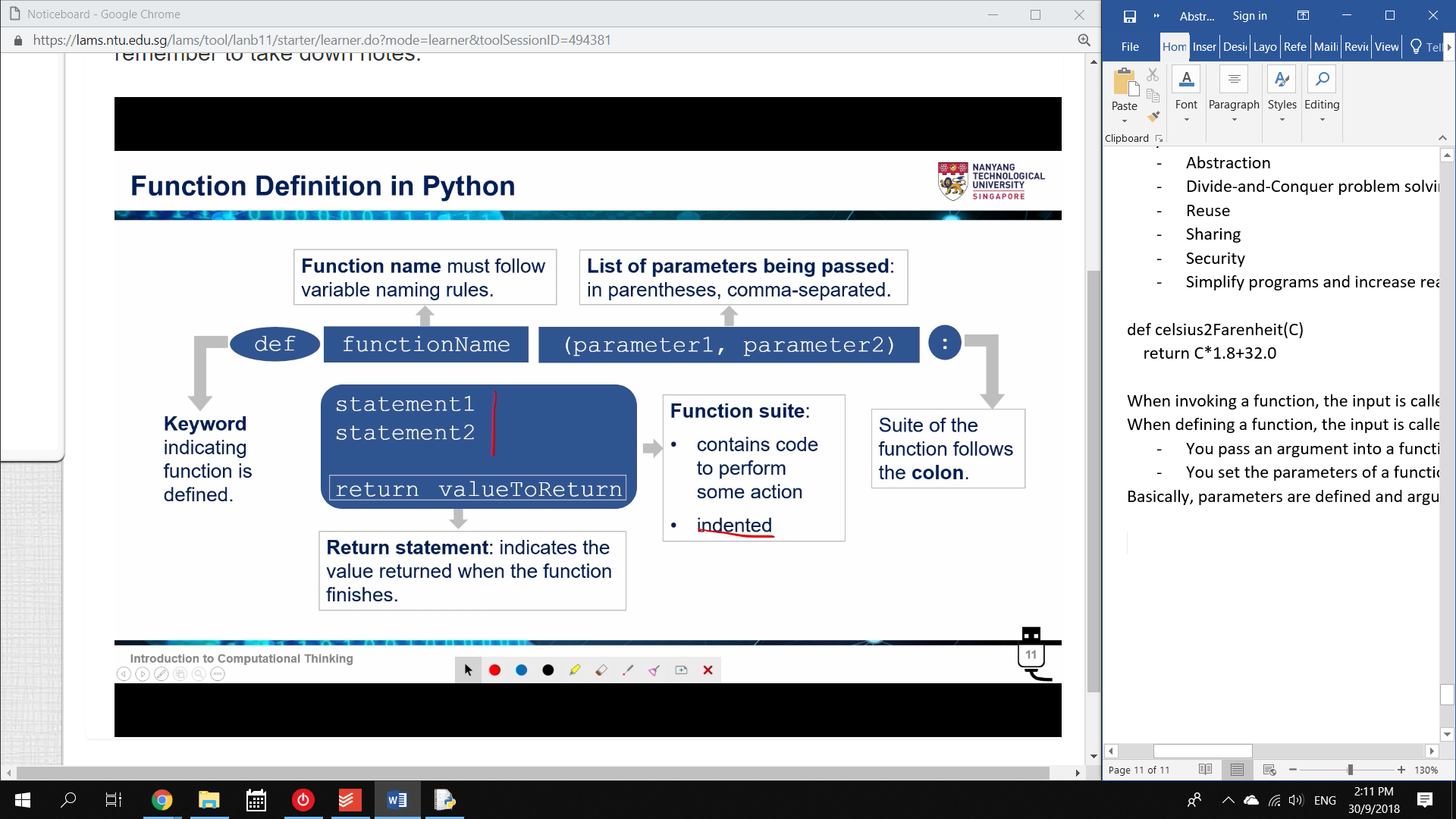
return C\*1.8+32.0

When invoking a function, the input is called an Argument

When defining a function, the input is called a Parameter.

* You pass an argument into a function.
* You set the parameters of a function.

Basically, parameters are defined and arguments are provided.



The return statement is not always required.

* If there is no return value, the function is called a Procedure.

How a function works:

1. Argument copied to parameter in Function

2. Control transfers to function

3. Expression is evaluated.

4. Result is Returned to invokation.

Principles of writing a function

Functions should:

* do ONE thing at a time
* be Readable
* be Reusable
* Complete: must apply to all possible cases
* Short

Procedures

Procedures do not have a return statement.

They are used to perform an action:

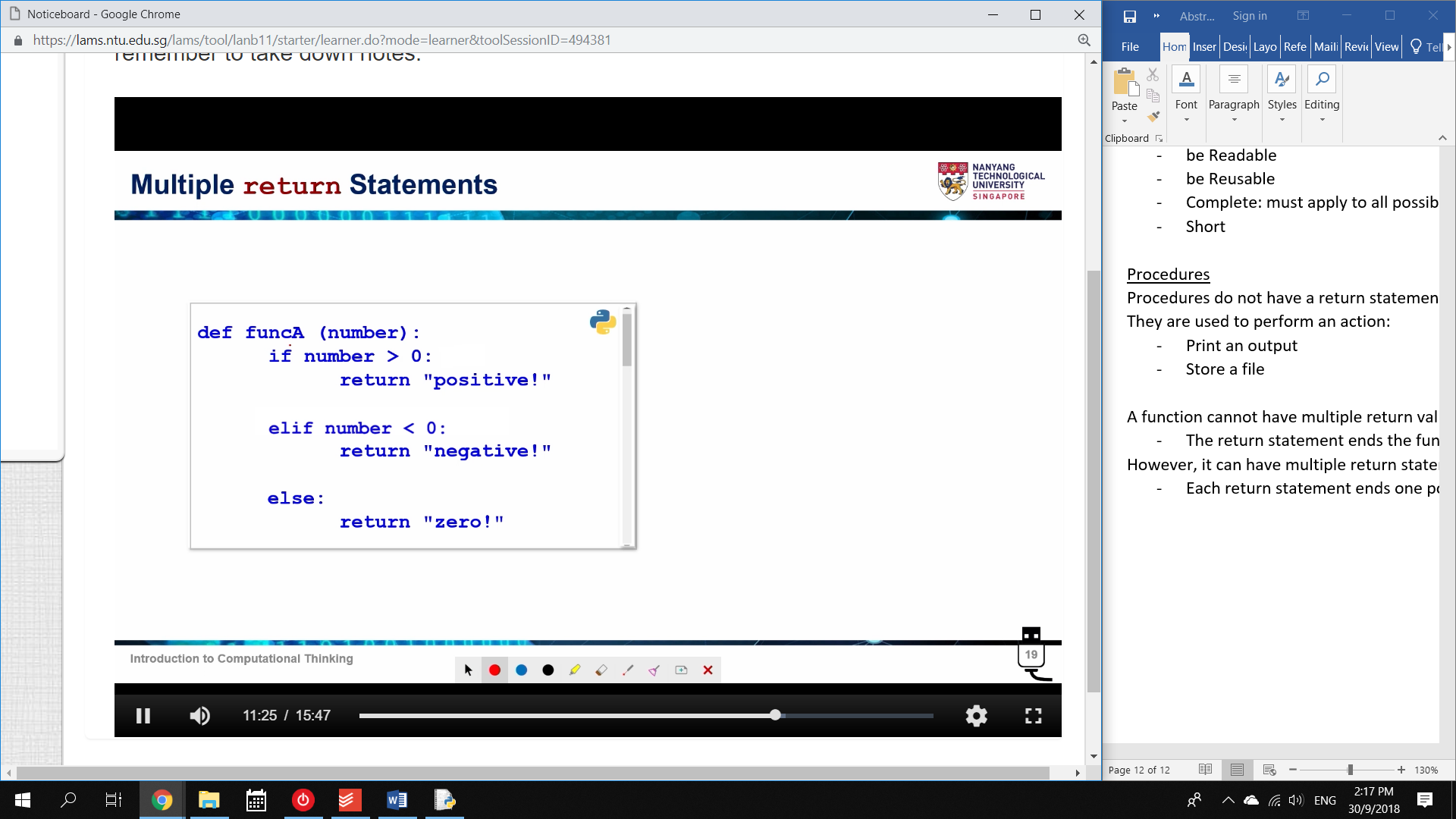
* Print an output
* Store a file

A function cannot have multiple return values.

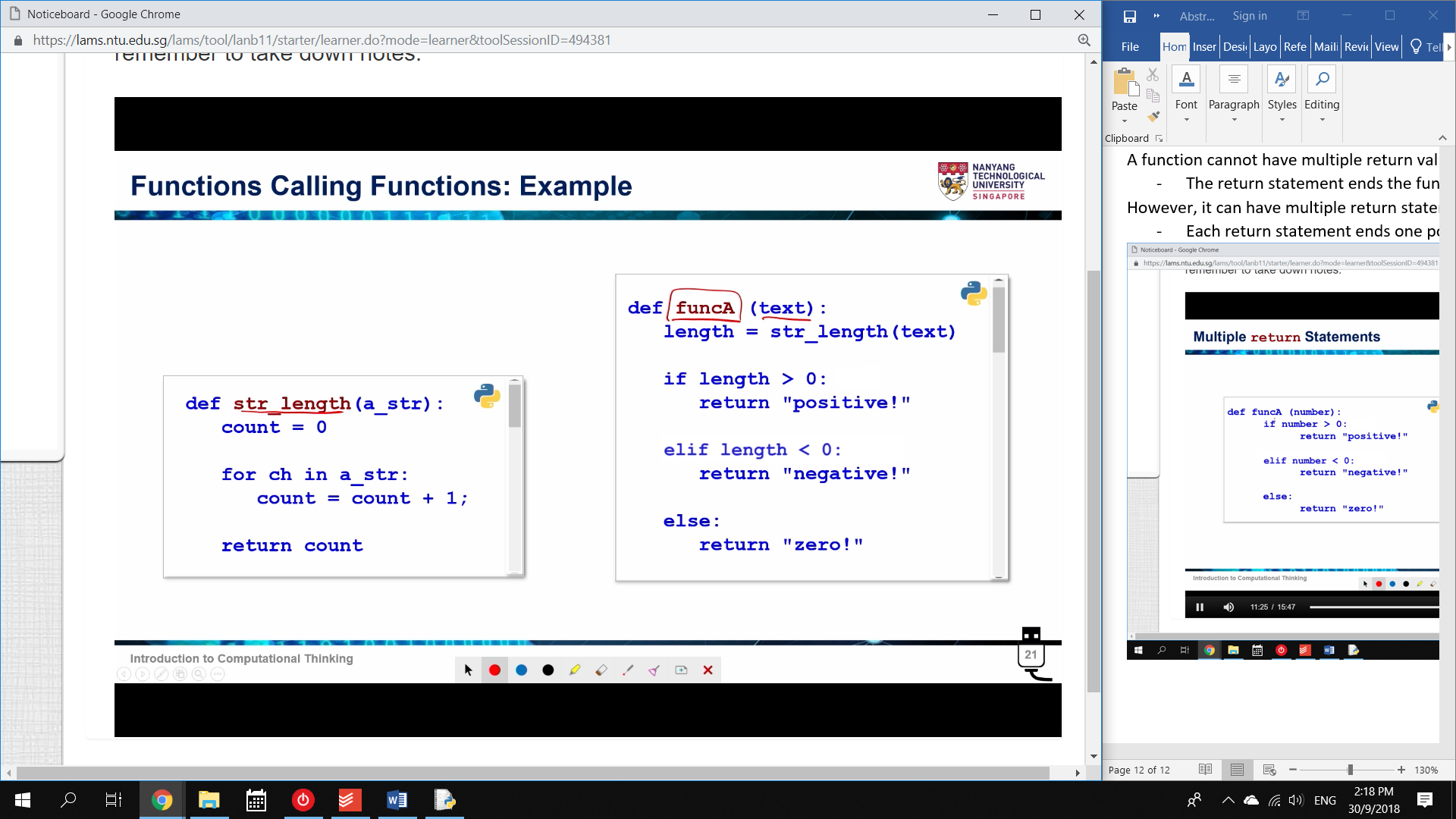
* The return statement ends the function.

However, it can have multiple return statements.

* Each return statement ends one possible branching path.



Functions calling Functions



A function can call another function.

The control is transferred to whichever function needs to evaluate at the moment.