CS1010E notes

Prof. Khoo Siau Cheng

Dr. Kok-Lim Low

Abstractions matter (hide the irrelevant details) no multitasking

Dr. Henry Chia

Dr. muhammad rizki

Lecture 1 Aug.12th

Algorithmic problem solving

input-algorithm-output

traits of algorithm: **exact/effective/general/must terminate**

e.g. gcd

Euclidean Algorithm

gcd(a,b)=gcd(b, rem(a,b)) (b>0)

= a (b=0)

lecture01.pdf

This is a declarative way

Paradigms:

Declaratively vs Imperatively

What to do vs How to do

Manipulate states vs stateless

for imperative paradigm: control flow

e.g. implementation for gcd

the flow is not obviously wrong

for this situation use a temp variable to store a/b in case of being changed before manipulation

Lec 2 Aug.19th

indentation maters in python

user———-implementor

function— high cohesion—do one thing and do one thing well

Scoping rule: var in functions is *local*

"***look from inside out***"

inside no, check if there's a outside one

Identifiers: case sensitive name of function and variables

letters, digits, underscores

can't start with digits

can't be a reserved word

don't use single ltr or standard names

Literal: fixed data of one type

integer/floating/boolean

expression:

e.g. 1+1 (got evaluated to 2 ; *an evaluation of the expression*)

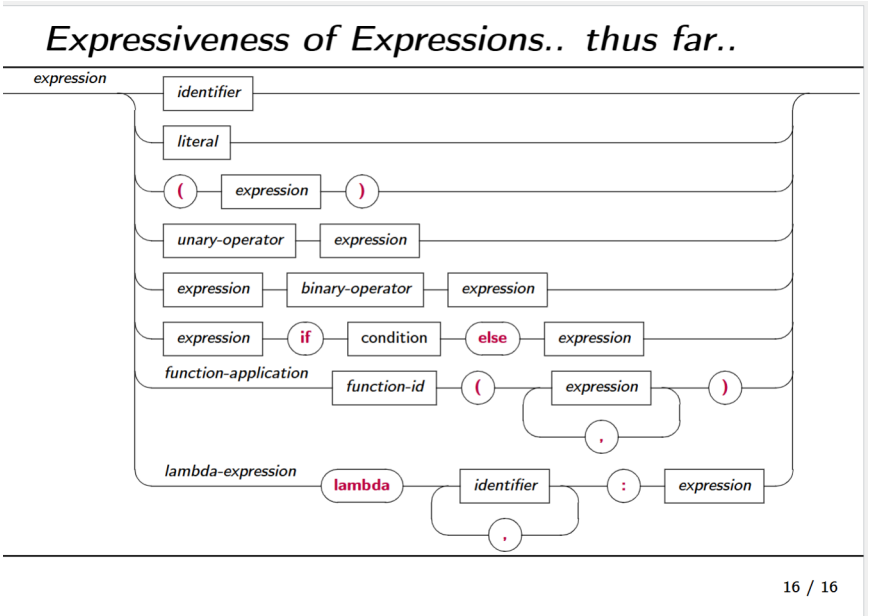
e.g. f(1)(evaluation of the function expression)

statement:

e.g. print(1+1) (A print *statement*)

e.g. return (return statement)

?



Arithmetic

- arithmetic operators : +-*/ ** //

Sidenote: // 整除　/ 正常除法

- evaluation

use parentheses () to group

<u>incremental coding</u>

Logical/ Boolean expression

short-circuit

A is False — A and B is also false

A is True — A or B is true

conditional expression(differs from statement!)

e.g. <expression> if <cond> else <expression>

tolerance: check whether the value is between a certain bound

first-order functions

higher-order functions

lambda expression

if we want independent values

use

```
lambda y: x
```

```
from math import sin
from math import pi

def g_sinc(f):
    return lambda x : 1.0 if abs(f(x) - 0.0) < 1e-15 \
        else sin(f(x)) / f(x)

def u_sinc(x):
    return g_sinc(lambda x : x)(x)
```

```
def n_sinc(x):
    return g_sinc(lambda x : pi * x)(x)
```

Why not just use g_sinc(pi*x)?

Reason: n_sinc(x) doesn't need to do the computation—it just passed an abstraction to g_sinc and let g_sinc do the calculation

READ THE QUESTIONS CAREFULLY!

Lec 3

ASCII table(128) : null—digit—uppercase—lowercase

'0' ≠ 0

character digits≠ integer digits

in most programming languages, 1.0 and 1 are considered as the same even their types are not the same

```
ord('.....')
```

```
chr(ordinal)
```

for conversion

e.g. use things like int('5') to convert the string into num

str(123) to convert the string into str

String: sequence of characters

single quotes or double quotes

empty string

```
''
```

index using

```
string[n]
```

[-1] means the last character or[len(s)-1]

join using

```
'asdhfkad'+ 'sdfjsk'

'cs1010e' * 2 = 'cs1010ecs1010e'
```

membership using

```
in
```

not operator— **check emptiness**

True when empty

```
not ''
True
```

Slicing

[start: stop]

[ )

[start: stop: step]

Special: [:] gives the sequence

[: : -1] gives the reversed sequence

start < step : forward slices w positive step

start > step: reverse slices w negative step

don't write confusing ones

Tuple (enclose in ()'s)

use

```
tuple()
```

to create a new tuple

or

```
tuple('dsh')
('d','s','h')
```

if there is only one elemnt in the tuple, you need to add a comma after the first element to indicate its a tuple

Range function

start-stop-step

Map function: applies a function to all elements

outcome: a map object which is iterable

map(function, elements)

sidenote: the ‘element’ can be a tuple or two tuples if the function can take in two values

print— ‘spy cameras’

Filter Function: select elements from a source iterable

```
filter(lambda x : x % 2 == 0, range(10))
```

Reduction:

```
sum
max
min
any
all
```

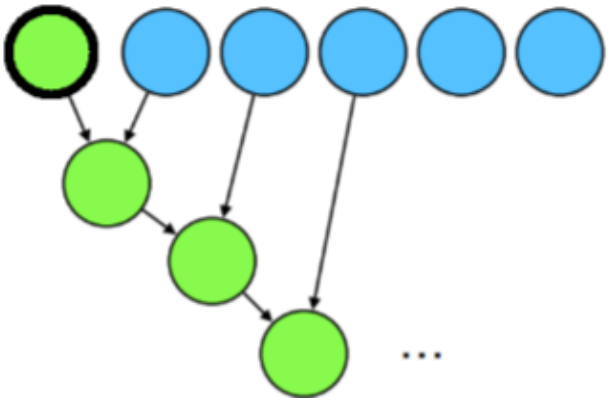
```
```

if cant use loop, can use filter to separate letters and numbers

Reduce Function:

remember:

```
from functools import reduce
```



e.g.

```
In [..]: reduce(lambda x,y : x * y, range(1, 10))
Out[..]: 362880
```

e.g.

```
def cumulative_sum(seq):
    ...:     ...:     t = lambda x : sum(seq[:x+1])
    ...:     ...:     return map(t, seq)

tuple(cumulative_sum(range(10)))
```

e.g.

```
def is_prime(n):
...: return n > 1 and \
...: all(map(lambda i: n % i != 0, range(2, n)))
```

map and filter: can only be iterated once

add initial condition in case that there’s an empty tuple

TUT1

gizemb

float division/ integer division

in python3 True = 1 False = 0

anything not 0 will be evaluated to True

Precedence:

BODMAS

Brackets first

Orders(powers and square roots)

Division and Multiplication

Addition and Subtraction

in python: from left to right

the integer produced by int( ) is not always smaller than the original one

e.g. negative numbers

conditional expression

Right associative: add panranthesis from right to left

use ()'s

arithmetic > comparison > not > and > or > conditional expression

in python, parameter has no declared types

be aware of the type

return: end of the function: anything behind will not be executed

higher order function: takes another function as a parameter

Lec WEEK 4

statements —-manipulating states

make a choice between declarative solutions and imperative ones

what to do: e.g. map —u dont car eabout how to implement the function

how to do:

program state: data stored in vars

change in content of a set of variable

Assignment statements:

e.g.

```
x = x + 1
```

evaluate RHS first, then assign value of RHS to LHS

```
(x,y) = (y,x)
```

to swap x,y (using tuple)

Definition of a function is a statement

Control structures- sequence

sequence

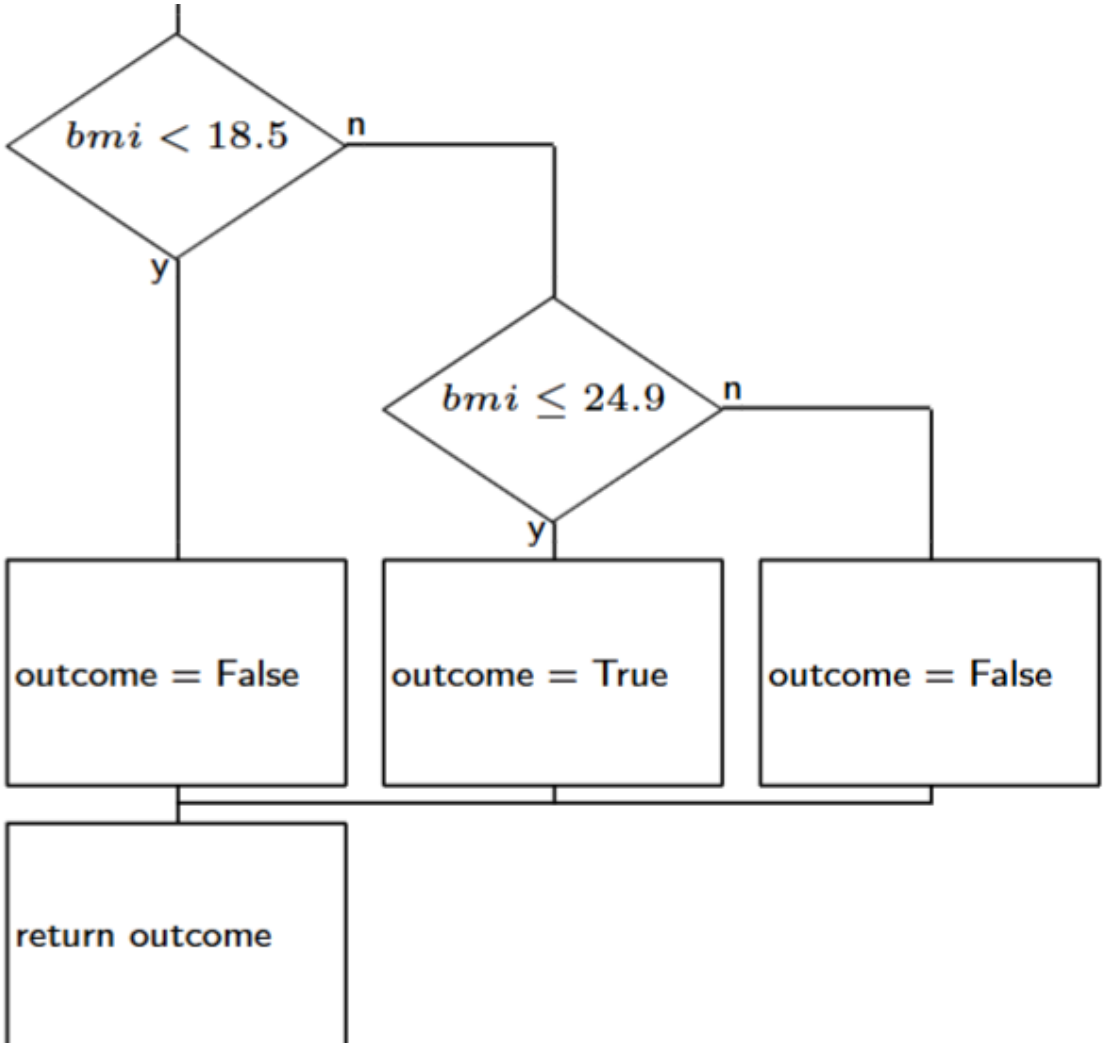
Statement Block

diamond: condition

rectangles :statements

selection:

e.g. Flow chart



draw the flow chart before writing your code

test the program using boundary values

repetition control

```
while condition:
```

```
for <var> in <iterable>
```

Count-Controlled Loop

sentinel-controlled loop

repeat until sentinel shows up

```
def is_prime(n):
    if (n < 2):
        return False
    for d in range(2,n):
        if n % d == 0:
            return False
    return True
```

"one entry, one exit"

```
def is_prime(n):
    i = 2
    while i < n and n % i != 0:
        i += 1
    return n == 1
```

input & output

e.g.

```
y = print(x)
123
```

the 123 is just a byproduct , not something evaluated

```
print(f'The number is {}')
```

e.g. round(<number>, 2)

sidenote:

```
from functools import reduce

def sum2(lst):
    # accumulator: (cumulative_sum, max_sum)
    def updater(acc, x):
        cum_sum = acc[0] + x
        max_sum = acc[1] if acc[1] > cum_sum else cum_sum
        print(str(x) + ',' + str(cum_sum) + '#')
        return (cum_sum, max_sum)

    final_acc = reduce(updater, lst, (0, lst[0]))
    print(str(final_acc[1]) + '#')
```

(alternative solution for as2 t3b)

TUT 2

e.g.

```
[2:] is the same as [2::]
```

```
s[]----syntax error
```

```
[::-1] # reverse the string
```

if step is negative—the default start will be at the end

lexicographical order:

- left to right
- winner detected when encountering different letter
- “anything” is greater than “nothing”

space is also an ascii character

beware that * applies different when dealing with integers and strings

we can also use * on tuples

e.g.

```
3 * (1,) = (1,1,1)
```

min(<str>) / max(<str>): ascii lexicographical order

```
min('scscscsc') = c
```

e.g.

```
tuple(range(5,6,-1))  # return ()
```

when using min(<tuple>), beware of different types like string and integers cannot be campared

filter: **elements which returns False are ruled out**

when there's a nested map/filter: read **from inside out**

if we want to transform an integer into iterable:

we can use

```
tuple(str(<number>))
```

e.g.

```
tuple(map(lambda x: int(x)**2,str(123456)))
# use string to turn it into an iterable and use int() to manipulate it again
```

e.g.

```
In [10]: reduce(lambda x,y: x +(y,y),(1,2,3,4),())
Out[10]: (1, 1, 2, 2, 3, 3, 4, 4)
```

```
#Name: Yang Xinjian
# NUSNET ID: E1710532
# Question number: 2
from functools import reduce
import math
def maclaurin_atan(x,k):
    general = lambda n: ((-1)**n) * (x**(2*n+1))/(2*n+1)
    poly = tuple(map(general, range(k+1)))
    return reduce(lambda x,y: x+y,poly)
```

```
#Name: Yang Xinjian
# NUSNET ID: E1710532
# Question number: 2
from functools import reduce
import math
def maclaurin_atan(x,k):
    general = lambda n: ((-1)**n) * (x**(2*n+1))/(2*n+1)
    poly = tuple(map(general, range(k+1)))
    return reduce(lambda x,y: x+y,poly)
```

Kadane’s algorithm: find the biggest sum of subarray

Lab week 5

recurrence relations

function activation stacks

Winding and Unwinding

- **Windup** phase:

  factorials are continually re-defined until $0!$

$$\begin{aligned} 5! &= \underline{5 \cdot 4!} \\ &= 5 \cdot \underline{4 \cdot 3!} \\ &= 5 \cdot 4 \cdot \underline{3 \cdot 2!} \\ &= 5 \cdot 4 \cdot 3 \cdot \underline{2 \cdot 1!} \\ &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot \underline{1 \cdot 0!} \\ &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot \underline{0!} \end{aligned}$$

- **Unwind** phase:

  substitute $0! = 1$ and unwind while substituting values for the factorials

$$\begin{aligned} & 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot \underline{1} \\ &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot \underline{1 \cdot (1)} \\ &= 5 \cdot 4 \cdot 3 \cdot \underline{2 \cdot (1)} \\ &= 5 \cdot 4 \cdot \underline{3 \cdot (2)} \\ &= 5 \cdot \underline{4 \cdot (6)} \\ &= \underline{5 \cdot (24)} \\ &= (120) \end{aligned}$$

recursive function:

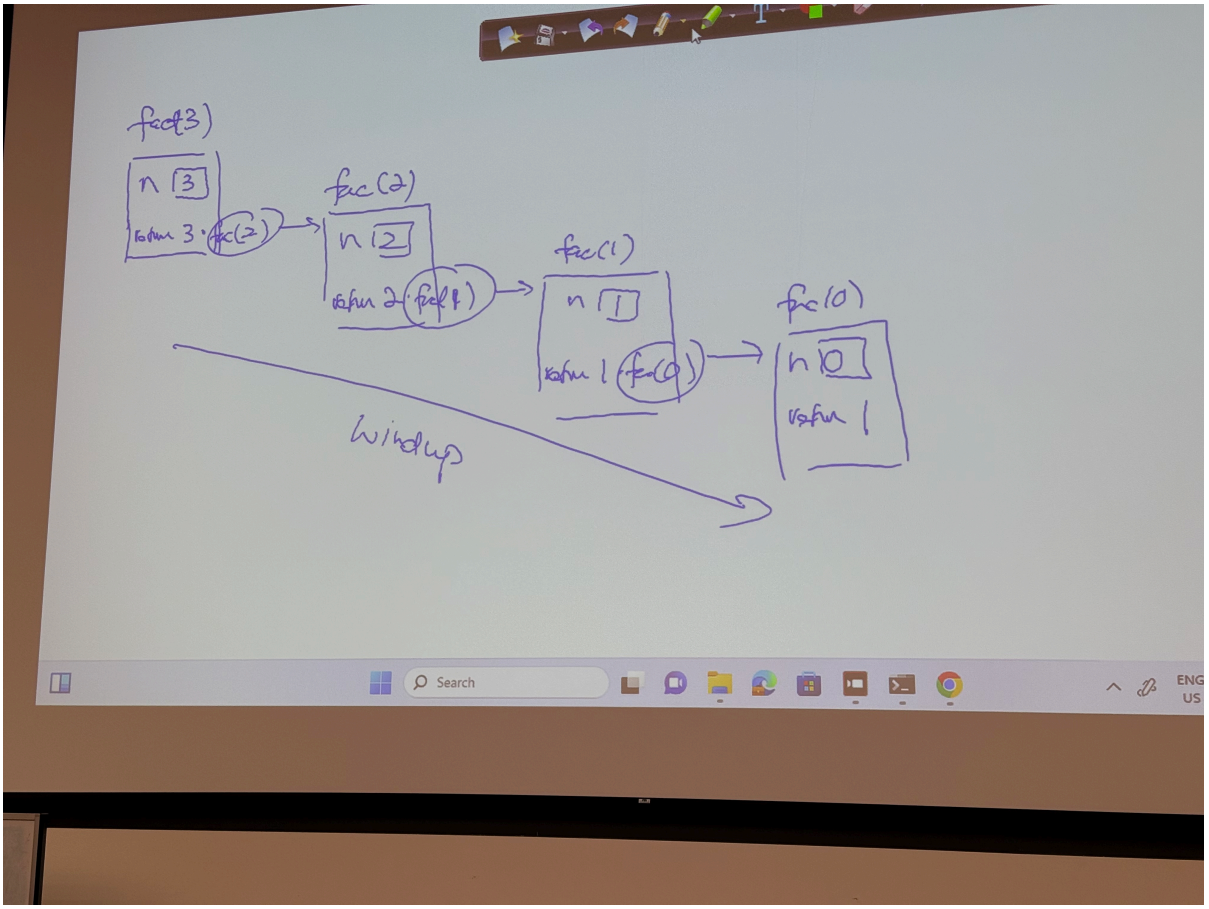
recursive case

base case

we also need a **selection construct**

```
def fac(n):
    if n == 0:
        return 1
    return  n * fac(n-1)
```

why is this a conventional way? `



windup then unwind

```
def fib(k):
  ...: return k if k <=1 else fib(k-1) + fib(k-2)
  ...:
```

recursive solution: elegant but time-consuming

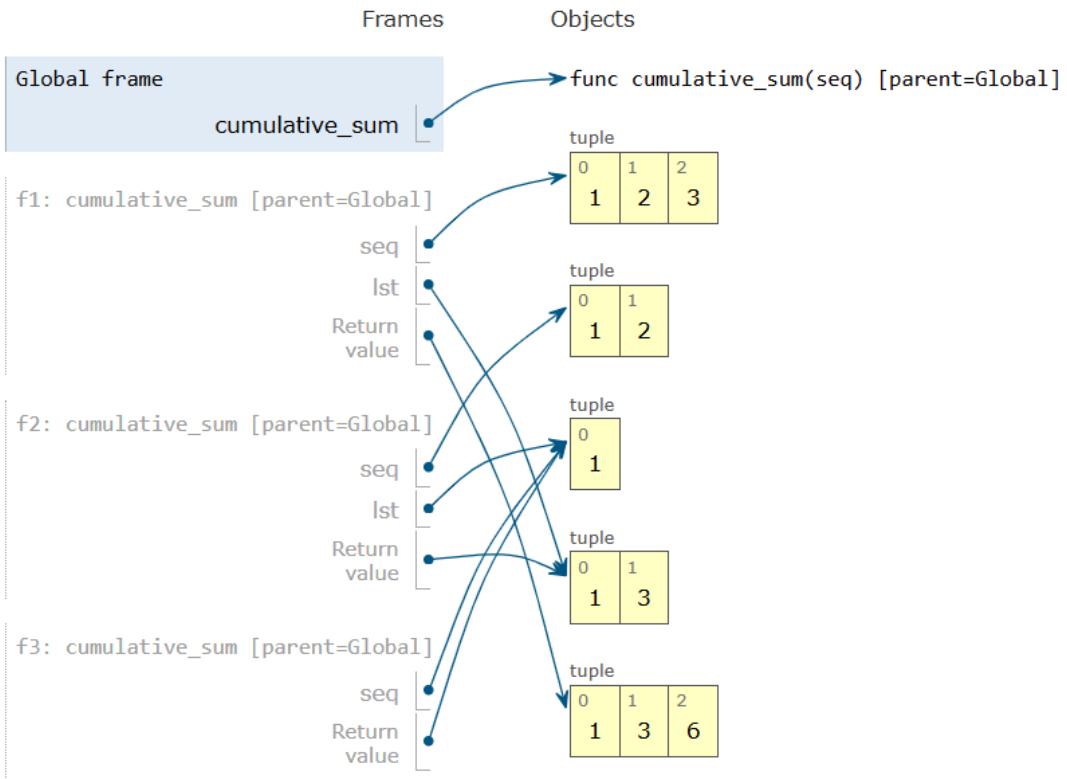
e.g.

```
def printDigits(n):
    if n > 0:
        print(n % 10)
        printDigits(n // 10)
# right to left
```

```
def printDigits(n):
    if n > 0:
        printDigits(n // 10)
        print(n % 10)
# left to right; loop cannot do
```

power of unwinding

```
def cumulative_sum(seq):
    if len(seq) <= 1:
        return seq
    lst = cumulative_sum(seq[:-1])   # set list to the tuple without the last ele
ment
    return lst + (lst[-1] + seq[-1],)
```



Towers of Hanoi

```
def tower(n, src, tmp, dst):
    if n > 0:
        towers(n-1, src, dst, tmp)
        print(f"Move disk {n} from {src} to {dst}")
        towers(n-1, tmp, src, dst)
```

One-layer thinking maxim: Don’t try to think recursively about a recursive process. — Concrete Abstractions

The way in which you would construct a recursive process is by wishful thinking. You have to believe. —— SICP

TUT 3

statements

Echoes: return values on the next line

some don’t have echoes because of no return value

e.g. print(x) in the definition of a function: is **form the return of print() function**

selection statements

looping statements

```
def print_stars(n):
    for rows in range(n,0,-1):
        toprint = ''
        for i in range(rows):
            toprint += '*'
        print(toprint)
```

```
print('*', end = '')
print('*')

#Output: **
```

remove things out of a tuple: make a new tuple and add the rest items in

quiz

```
# Name: Yang Xinjian
# NUSNET ID: E1710532
# Question 9

def diff_type_idx(tup):
    indextuple = ()
    for i in range(len(tup)-1):
        if type(tup[i]) != type(tup[i+1]):
            indextuple = indextuple + ((i,i+1),)
    return indextuple
```

A little bit of thought: when tackling a big problem— always try to break it down into smaller ones—- u never know whether u gonna use these small functions in the future...

```
def encode_R(word):

    def get_index(character,index):
        uppercase = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        if uppercase[index] == character:
            return index
```

```
        else: return get_index(character,index+1)

    if word == '': return ''

    chr1 = word[0]
    if chr1 == ' ': str1 = '99'
    else:
        index = get_index(chr1,0)
        if index <= 9: str1 = '0' + str(index)
        else: str1 = str(index)

    return str1 + encode_R(word[1:])
```

p.s. good example of how to divide and conquer; and how to use recursive method to replace iteratives

things to note:

```
a, b = 1, 2
b, a = a, b
print(a, b)
```

- The right-hand side is **evaluated first**, producing the tuple `(1, 2)`.
- Then Python unpacks that tuple into the left-hand side variables `b` and `a`.

create a tuple and then unpack it

Sidenote: the terminated the code in

```
return  # terminate the code in def
```

in recursion— the print statement doesn't need to be 'unwinded'

Conway sequence:

P(n) = P(P(n-1)) + P(n - P(n-1))

Hofstadter Female and Male Sequences:

F(n) = n - M(F(n-1))

M(n) = n - F(M(n-1))

Stern-Brocot Construction:

e.g.

```
0: [(0,1),                    (1,1)]
1: [(0,1),       (1,2),       (1,1)]
2: [(0,1), (1,3), (1,2), (2,3), (1,1)]
```

Lec 6: Objects and Mutable Sequences

every value in python is a subject

“pass by reference”

```
id(<object>) # obtain the address of the object
```

when we use

```
x # refers to @..... and use the value inside @.....
```

after u create a immutable object , u can never change the content under the same address

```
In [..]: x = 1000
In [..]: y = 1000
In [..]: x == y
Out[..]: True
In [..]: x is y
Out[..]: False
```

“Integer caching” [-5,256]

small integers get a fixed address

```
is # to check whether 2 varibales refer to the same object
```

List

primary effects vs side effects:

side effects: if there’s another effect other than the primary effect

side effects:

when two lists refer to the same list object— when one changes, the other also changes

Manipulating the states:

```
def c_sum(list):
    for i in range(1,len(lst)):
        lst[i] = lst[i] + lst[i-1]
    return lst
```

every time u call

```
c_sum(lst)
```

the state will be changed, which is what we dont want

Pass by reference

Heap memory:

stack:

?

to avoid side effect: make a copy of the list u want to modify

List comprehension:

```
[<expression> for <var> in <iterator> if ...]
```

```
list(<expression> for <var> in <iterator> if ...)
```

```
tuple(<expression> for <var> in <iterator> if ...)
```

List & Tuples:

list to store large homogeneous elements of the same type

tuple to store small group of heterogeneous data

Sidenote: avoid calling functions(especially recursive) every time if the return value can be recorded in a var

TUT 4

```
def ne(x,y):
    if x == 0 or y == 0 :
        return 1
    else: return ne(x-1,y) + ne(x,y-1)
```

(condition: only N and E, using recursion)

if no base case: stack overflow

```
def digit_sum(n):
    if n == 0: return 0
    else: return n%10 + digit_sum(n//10)

def final_sum(n):
    if 0 <= n <= 9: return n
    else:
        return final_sum(digit_sum(n))

def final_sum(n):
    while n > 9:
        n = digit_sum(n)
    return n
```

choose between while loop and for loop : depends on whether u know the times it will be iterated

```
lst1 = ['a', 'b', 'c']
lst2 = lst1[:]    # Creates a new list
```

```
lst1 == lst2      # correct
lst1 is lst2      # wrong
```

```
lst = [1, 2, 3]
lst[-1:] = []
# removes the whole slice:
# lst is [1,2]
```

```
lst = [1, 2, 3]
lst[-1] = []
# replace the last element with []
# lst is [1,2,[]]
```

- Using `[...]` → refers to a single element. Replacement changes *what is stored*.
- Using `[...] : [...]` (slice) → refers to a *range of elements*. Replacement can insert, delete, or replace multiple elements at once.

```
# two different methods of inserting a value in the list
    lst = lst[:idx] + [elem] + lst[idx:]

    lst[idx:idx] = [elem]
```

pay attention to the modifications of the list!