

INFO-H-100 – Informatique – Prof. Th. Massart
1^{ère} année du grade de Bachelier en Sciences de l'Ingénieur
Interrogation de juin

Prénom : _____ Nom : _____ Matricule : _____

Q1 (/2)	Q2 (/2)	Q3 (/3)	Q4 (/5)	Q5 (/4)	Q6 (/4)	Total (/20)

6 questions, 10 feuilles.

Remarques préliminaires à lire attentivement !

- On vous demande de **répondre** aux questions **sur les feuilles de l'énoncé**. Si nécessaire, vous pouvez continuer la réponse sur le verso de la page.
- Des feuilles de brouillon ainsi que l'aide-mémoire vous sont fournis en fin d'énoncé. S'il vous faut plus de feuilles de brouillon, demandez-en.
- **Les feuilles d'énoncé doivent rester agrafées**. Vous pouvez détacher uniquement les feuilles de brouillon et l'aide-mémoire.
- N'oubliez pas d'inscrire votre nom, prénom et numéro de matricule sur **chaque** feuille (y compris celle-ci).
- Vous disposez de **trois heures** et vous ne pouvez pas utiliser de notes.
- Si du code vous est demandé,
 - la réponse à la question doit comprendre le code *Python* structuré et conforme aux règles de bonne pratique et conventions,
 - sauf mention contraire, vous ne pouvez utiliser aucune fonction de librairies (pas d'`import`)
 - veuillez à découper votre réponse en fonctions de manière pertinente
 - veuillez à utiliser des structures de données appropriées.
- Veuillez à écrire lisiblement et n'hésitez pas à commenter votre code si ça peut le rendre plus clair.

Question 1 - Run Time (2 points)

Expliquez comment fonctionne à l'exécution le "run time" d'un programme Python. En particulier, expliquer, grâce à des diagrammes d'état, comment le programme récursif ci-dessous fonctionne.

```
def evaluate(v):
    if type(v) is not list:
        res= v
    elif v[1]==' ':
        res = evaluate(v[0]) + evaluate(v[2])
    elif v[1]=='-':
        res = evaluate(v[0]) - evaluate(v[2])
    elif v[1] == '*':
        res = evaluate(v[0]) * evaluate(v[2])
    elif v[1]=='/':
        res = evaluate(v[0]) / evaluate(v[2])
    else:
        res = None # error
    return res

exemple_exp = [3, '+', [4, '*', 5]]
res= evaluate(exemple_exp)
```


Prénom :

Nom :

Matricule :

Question 2 - Fonctionnement de Python 2 (2 points)

Expliquez le mode de fonctionnement d'un compilateur et d'un interpréteur. Python de base est-il interprété ou compilé ? Expliquez quels en sont les avantages et les inconvénients pour ce langage.

Question 3 - Complexité (3 points)

En supposant que l'exécution se passe sans erreur, donnez la complexité moyenne et maximale de chaque instruction du code suivant (en fonction des input). **Expliquez** pourquoi nous avons ces complexités (une réponse sans explication n'a pas de valeur).

```
t = list(input())

d = dict(zip(range(len(t)), t))
# aide: si t = ['b', 'o', 'n', 'j', 'o', 'u', 'r']
#       d = {0: 'b', 1: 'o', 2: 'n', 3: 'j', 4: 'o', 5: 'u', 6: 'r'}

i=int(input())

x=t[i]

t.append('!')

d[len(t)] = '!'

print(d[len(t)])

del t[-1]

del d[len(t)+1]

j = 1
while j < i:
    print(j)
    j = 2*j

j = 2
while j < i:
    print(j)
    j = j*j
```


Question 4 - Manipulation de fichiers et de données (5 points)

Écrivez une fonction `fusion_h(fname_1, fname_2)` qui **affiche** (la fonction ne renvoie donc rien) le contenu de deux fichiers en concaténant les lignes correspondantes. Chaque ligne affichée est obtenue en mettant bout-à-bout les lignes correspondantes des deux fichiers et en les séparant par un espace.

Exemple :**Fichier** `f1.txt`:

```
Ceci est un fichier
qui contient un texte
court
```

```
Le but est d'illustrer
le fonctionnement du programme
```

Fichier `f2.txt`:

```
Un fichier
quelconque et
sans importance
```

Résultat de `fusion_h("f1.txt", "f2.txt")`:

```
Ceci est un fichier Un fichier
qui contient un texte quelconque et
court
sans importance
Le but est d'illustrer
le fonctionnement du programme
```

Note : Vous pouvez utiliser la fonction `st.rstrip()` sur une chaîne de caractères `st` pour retirer les “blancs” (espaces, sauts de lignes, tabulation...) qui se trouvent à la fin de cette chaîne (“à droite”).

Question 5 - Tri (4 points)

On vous demande d'écrire une fonction `sorted_rands(n)` qui génère une liste *triée* de n nombres aléatoires dans l'intervalle $[0, 1[$. La fonction doit impérativement construire la liste *au fur et à mesure de la génération* des nouvelles valeurs et non pas en deux étapes (génération d'une liste de nombres aléatoires, puis tri de cette liste générée). Pratiquement, nous vous suggérons d'adapter la méthode de tri par insertion.

Exemple :

```
>>> sorted_rands(5)
[0.1525017040560619, 0.18414999412515853, 0.323816213456277, 0.5775189595078108, 0.9786976725577308]
```

Notes :

- Utilisez la fonction `random.random()` de la librairie `random` qui génère une valeur aléatoire dans l'intervalle $[0, 1[$.
- Vous ne pouvez pas utiliser de fonction de tri existante du type `sort` ou `sorted`.

Question 6 - Récursivité (4 points)

On vous demande d'écrire une fonction `count_divisibles(l1ist, div)` qui renvoie le nombre d'éléments de la liste `l1ist` qui sont divisibles par la valeur entière `div`.

Le paramètre `l1ist` est une liste dont chaque élément peut être une valeur entière ou une sous-liste. Vous pouvez considérer que le format des paramètres `l1ist` et `div` transmis sont valides ; vous ne devez pas le vérifier.

Exemple :

```
>>> mylist = [1, [ [4, 3], 4 ], 2, [9, 3] ]
>>> count_divisibles(mylist, 2)
3
```


Prénom :

Nom :

Matricule :

Prénom : Nom : Matricule :

Prénom : Nom : Matricule :

Base Types

integer, float, boolean, string

```
int 783 0 -192
float 9.23 0.0 -1.7e-6
bool True False
str "One\nTwo" 'I\'m'
```

↑
immutable,
ordered sequence of chars

new line
multiline
escaped
tab char

Container Types

- ordered sequence, fast index access, repeatable values
- no *a priori* order, unique key, fast key access ; keys = base types or tuples

```
list [1,5,9] ["x",11,8.9] ["word"] []
tuple (1,5,9) 11,"y",7.4 ("word",) ()
dict {"key":"value"} {}
dictionary {1:"one",3:"three",2:"two",3.14:"pi"}
key/value associations
set {"key1","key2"} {1,9,3,0} set()
```

expression with just comas
as an ordered sequence of chars

Identifiers

for variables, functions, modules, classes... names

a..zA..Z followed by **a..zA..Z_0..9**

- diacritics allowed but should be avoided
- language keywords forbidden
- lower/UPPER case discrimination

☺ **a toto x7 y_max BigOne**
☹ **8y and**

Variables assignment

```
x = 1.2+8+sin(0)
y,z,r = 9.2,-7.6,"bad"
```

↑
value or computed expression
variable name (identifier)

variables names
container with several values (here a tuple)

increment
decrement
x+=3
x-=2

x=None « undefined » constant value

Conversions

type (expression)

```
int("15") can specify integer number base in 2nd parameter
int(15.56) truncate decimal part (round(15.56) for rounded integer)
float("-11.24e8")
str(78.3) and for litteral representation → repr("Text")
see other side for string forming allowing finer control
bool → use comparators (with ==, !=, <, >, ...), logical boolean result
```

use each element from sequence
use each element from sequence

```
list("abc") → ['a','b','c']
dict([(3,"three"),(1,"one")]) → {1:'one',3:'three'}
set(["one","two"]) → {'one','two'}
```

joining string
sequence of strings

```
":".join(['toto','12','pswd']) → 'toto:12:pswd'
"words with spaces".split() → ['words','with','spaces']
"1,4,8,2".split(",") → ['1','4','8','2']
splitting string
```

Sequences indexing

for lists, tuples, strings, ...

len(lst) → 6

individual access to items via [index]

```
lst[1] → 67
lst[0] → 11 first one
lst[-2] → 42
lst[-1] → 1968 last one
```

access to sub-sequences via [start slice:end slice:step]

```
lst[1:3] → [67,"abc"]
lst[-3:-1] → [3.14,42]
lst[:3] → [11,67,"abc"]
lst[4:] → [42,1968]
```

Missing slice indication → from start / up to end.

On mutable sequences, usable to remove **del lst[3:5]** and to modify with assignment **lst[1:4]=['hop',9]**

Boolean Logic

Comparators: < > <= >= == !=

a and b logical and
both simultaneously

a or b logical or
one or other or both

not a logical not

True true constant value

False false constant value

Statements Blocks

```
parent statement:
statements block 1...
parent statement:
statements block 2...
next statement after block 1
```

indentation !

Conditional Statement

statements block executed only if a condition is true

if logical expression:
statements block

can go with several elif, elif... and only one final else, example :

```
if x==42:
    # block if logical expression x==42 is true
    print("real truth")
elif x>0:
    # else block if logical expression x>0 is true
    print("be positive")
elif bFinished:
    # else block if boolean variable bFinished is true
    print("how, finished")
else:
    # else block for other cases
    print("when it's not")
```

Maths

floating point numbers... approximated values!

angles in radians

Operators: + - * / // % **
× ÷ ↑ ↑ a^b
integer ÷ ÷ remainder

```
from math import sin,pi...
sin(pi/4) → 0.707...
cos(2*pi/3) → -0.4999...
acos(0.5) → 1.0471...
sqrt(81) → 9.0
log(e**2) → 2.0 etc. (cf doc)
```

(1+5.3)*2 → 12.6

abs(-3.2) → 3.2

round(3.57,1) → 3.6

statements block executed as long as condition is true **Conditional loop statement**

while logical expression:

→ statements block

s = 0
i = 1 } initializations **before** the loop

condition with at least one variable value (here **i**)

while i <= 100:

statement executed as long as $i \leq 100$

s = s + i2**

i = i + 1 } make condition variable change

print("sum:", s) } computed result after the loop
be careful of infinite loops!

$$s = \sum_{i=1}^{i=100} i^2$$

statements block executed for each item of a container or iterator **Iterative loop statement**

for variable **in** sequence:

→ statements block

Go over sequence's values

s = "Some text"
cnt = 0 } initializations **before** the loop

loop variable, value managed by **for** statement

for c in s:

if c == "e":

cnt = cnt + 1

print("found", cnt, "'e'")

Count number of **e** in the string

loop on dict/set = loop on sequence of keys
use slices to go over a subset of the sequence

Go over sequence's index

□ modify item at index

□ access items around index (before/after)

lst = [11, 18, 9, 12, 23, 4, 17]

lost = []

for idx in range(len(lst)):

val = lst[idx]

if val > 15:

lost.append(val)

lst[idx] = 15

print("modif:", lst, "-lost:", lost)

Limit values greater than 15, memorization of lost values.

Go simultaneously over sequence's index and values:

for idx, val in enumerate(lst):

print("v=", 3, "cm :", x, ", ", y+4) **Display / Input**

items to display: literal values, variables, expressions

print options:

□ **sep=" "** (items separator, default space)

□ **end="\n"** (end of print, default new line)

□ **file=f** (print to file, default standard output)

s = input("Instructions: ")

input always returns a **string**, convert it to required type
(cf boxed **Conversions** on the other side).

len(c) → items count

min(c) **max(c)** **sum(c)**

sorted(c) → sorted copy

val in c → boolean, membership operator **in** (absence **not in**)

enumerate(c) → iterator on (index, value)

Special for **sequence containers** (lists, tuples, strings):

reversed(c) → reverse iterator

c*5 → duplicate

c+c2 → concatenate

c.index(val) → position

c.count(val) → events count

Operations on containers

Note: For dictionaries and set, these operations use **keys**.

modify original list

lst.append(item)

add item at end

lst.extend(seq)

add sequence of items at end

lst.insert(idx, val)

insert item at index

lst.remove(val)

remove first item with value

lst.pop(idx)

remove item at index and return its value

lst.sort() **lst.reverse()**

sort / reverse list in place

Operations on lists

Operations on dictionaries

d[key]=value **d.clear()**

d[key]→value **del d[clé]**

d.update(d2) { update/add

associations

d.keys()

views on keys, values

d.values()

views on keys, values

d.items()

associations

d.pop(clé)

Operations on sets

Operators:

| → union (vertical bar char)

& → intersection

- ^ → difference/symmetric diff

< > >= → inclusion relations

s.update(s2)

s.add(key) **s.remove(key)**

s.discard(key)

frequently used in **for** iterative loops

Generator of int sequences

default 0, not included
range([start,]stop[,step])

range(5) → 0 1 2 3 4

range(3, 8) → 3 4 5 6 7

range(2, 12, 3) → 2 5 8 11

range returns a « generator », converts it to list to see the values, example:

print(list(range(4)))

function name (identifier)

Function definition

named parameters

def fctname(p_x, p_y, p_z):

"""documentation"""

→ # statements block, res computation, etc.

return res ← result value of the call.

parameters and all of this block only exist in the block and during the function call ("black box")

if no computed result to return: **return None**

Function call

r = fctname(3, i+2, 2*i)

one argument per parameter

retrieve returned result (if necessary)

storing data on disk, and reading it back

Files

f = open("fil.txt", "w", encoding="utf8")

file variable

name of file

opening mode

encoding of

for operations

on disk

(+path...)

files:

cf functions in modules **os** and **os.path**

□ 'r' read

□ 'w' write

utf8 ascii

□ 'a' append...

latin1 ...

...

writing

f.write("hello")

text file → read/write only
strings, convert from/to required type.

f.close() don't forget to close file after use

Pythonic automatic close: **with open(...) as f:**

very common: iterative loop reading lines of a text file

for line in f:

→ # line processing block

empty string if end of file

reading

s = f.read(4)

if char count not specified, read whole file

s = f.readline()

read next line

Strings formatting

formatting directives

values to format

"model {} {} {}".format(x, y, r) → **str**

"{selection:formatting!conversion}"

□ Selection:

2

x

0.nom

4[key]

0[2]

Examples

"{:+2.3f}".format(45.7273)

→ **'+45.727'**

"{1:>10s}".format(8, "toto")

→ **'toto'**

"{!r}".format("I'm")

→ **'"I\'m"'**

□ Formatting:

fillchar alignment sign minwidth.precision-maxwidth type

<> ^ = + - space

0 at start for filling with **0**

integer: **b** binary, **c** char, **d** decimal (default), **o** octal, **x** or **X** hexa...

float: **e** or **E** exponential, **f** or **F** fixed point, **g** or **G** appropriate (default),

% percent

string: **s**...

□ Conversion: **s** (readable text) or **r** (literal representation)