Lesson 1: Sample Exam Questions

Created: 2024-05-27T01:25:53.761254+10:00

Python 3 Cheat Sheet

Question 1

Complete the code in the function that, given a list L of random non negative whole numbers, decomposes L into a list R of increasing sequences and with consecutive duplicates removed.Here are some execution examples: >>> f(0, 0, 10) Here is L: [] The decomposition of L into increasing sequences, with consecutive duplicates removed, is: [] >>> f(0, 1, 10) Here is L: [6] The decomposition of L into increasing sequences, with consecutive duplicates removed, is: [[6]] >>> f(0, 2, 10) Here is L: [6, 6] The decomposition of L into increasing sequences, with consecutive duplicates removed, is: [[6]] >>> f(0, 3, 10) Here is L: [6, 6, 0] The decomposition of L into increasing sequences, with consecutive duplicates removed, is: [[6], [0]] >>> f(0, 4, 10) Here is L: [6, 6, 0, 4] The decomposition of L into increasing sequences, with consecutive duplicates removed, is: [[6], [0, 4]] >>> f(0, 5, 10) Here is L: [6, 6, 0, 4, 8] The decomposition of L into increasing sequences, with consecutive duplicates removed, is: [[6], [0, 4, 8]] >>> f(0, 6, 10) Here is L: [6, 6, 0, 4, 8, 7] The decomposition of L into increasing sequences, with consecutive duplicates removed, is: [[6], [0, 4, 8], [7]] >>> f(0, 7, 10) Here is L: [6, 6, 0, 4, 8, 7, 6] The decomposition of L into increasing sequences, with consecutive duplicates removed, is: [[6], [0, 4, 8], [7], [6]] >>> f(3, 10, 6) Here is L: [1, 4, 4, 1, 2, 4, 3, 5, 4, 0] The decomposition of L into increasing sequences, with consecutive duplicates removed, is: [[1, 4], [1, 2, 4], [3, 5], [4], [0]] >>> f(3, 15, 8) Here is L: [3, 8, 2, 5, 7, 1, 0, 7, 4, 8, 3, 3, 7, 8, 8] The decomposition of L into increasing sequences, with consecutive duplicates removed, is: [[3, 8], [2, 5, 7], [1], [0, 7], [4, 8], [3, 7, 8]]

Question 2

Write a function that accepts a strictly positive integer and displays its binary representation as well as the number of times the value 1 appears in its binary representation. You might find the function bin() useful.Here are some execution examples: >>> f(1) 1 in binary reads as: 1. Only one bit is set to 1 in the binary representation of 1. >>> f(2) 2 in binary reads as: 10. Only one bit is set to 1 in the binary representation of 2. >>> f(3) 3 in binary reads as: 11. 2 bits are set to 1 in the binary representation of 3. >>> f(7) 7 in binary reads as: 111. 3 bits are set to 1 in the binary representation of 7. >>> f(2314) 2314 in binary reads as: 100100001010. 4 bits are set to 1 in the binary representation of 2314. >>> f(9871) 9871 in binary reads as: 10011010001111. 8 bits are set to 1 in the binary representation of 9871.

Question 3

Write a function that accepts a strictly positive integer greater or equal to 2 and "not too large" and displays its decomposition into prime factors.Here are some execution examples: >>> f(2) The decomposition of 2 into prime factors reads: 2 = 2 >>> f(3) The decomposition of 3 into prime factors reads: 3 = 3 >>> f(4) The decomposition of 4 into prime factors reads: 4 = 2^2 >>> f(5) The decomposition of 5 into prime factors reads: 5 = 5 >>> f(6) The decomposition of 6 into prime factors reads: 6 = 2 x 3 >>> f(8) The decomposition of 8 into prime factors reads: 8 = 2^3 >>> f(10) The decomposition of 10 into prime factors reads: 10 = 2 x 5 >>> f(15) The decomposition of 15 into prime factors reads: 15 = 3 x 5 >>> f(100) The decomposition of 100 into prime factors reads: 100 = 2^2 x 5^2 >>> f(5432) The decomposition of 5432 into prime factors reads: 5432 = 2^3 x 7 x 97 >>> f(45103) The decomposition of 45103 into prime factors reads: 45103 = 23 x 37 x 53 >>> f(45100) The decomposition of 45100 into prime factors reads: 45100 = 2^2 x 5^2 x 11 x 41

Question 4

Will be tested with a at least equal to 2 and b at most equal to 10_000_000.Here are some execution examples: >>> f(2, 2) There is a unique prime number beween 2 and 2. >>> f(2, 3) There are 2 prime numbers between 2 and 3. >>> f(2, 5) There are 3 prime numbers between 2 and 5. >>> f(4, 4) There is no prime number beween 4 and 4. >>> f(14, 16) There is no prime number beween 14 and 16. >>> f(3, 20) There are 7 prime numbers between 3 and 20. >>> f(100, 800) There are 114 prime numbers between 100 and 800. >>> f(123, 456789) There are 38194 prime numbers between 123 and 456789.

Question 5

Write a function that accepts a year between 1913 and 2013 inclusive and displays the maximum inflation during that year and the month(s) in which it was achieved.You might find the reader() function of the csv module useful, but you can also use the split() method of the str class.Make use of the attached cpiai.csv file.Here are some execution examples: >>> f(1914) In 1914, maximum inflation was: 2.0 It was achieved in the following months: Aug >>> f(1922) In 1922, maximum inflation was: 0.6 It was achieved in the following months: Jul, Oct, Nov, Dec >>> f(1995) In 1995, maximum inflation was: 0.4 It was achieved in the following months: Jan, Feb >>> f(2013) In 2013, maximum inflation was: 0.82 It was achieved in the following months: Feb

Question 6

You might find the zip() function useful, though you can also do without it.Here are some execution examples: >>> f(0, 2, 2) Here is the square: 1 1 0 1 It is not a good square because it contains duplicates, namely: 1 >>> f(0, 3, 5) Here is the square: 3 3 0 2 4 3 3 2 3 It is not a good square because it contains duplicates, namely: 2 3 >>> f(0, 6, 50) Here is the square: 24 48 26 2 16 32 31 25 19 30 22 37 13 32 8 18 8 48 6 39 16 34 45 38 9 19 6 46 4 43 21 30 35 6 22 27 It is not a good square because it contains duplicates, namely: 6 8 16 19 22 30 32 48 >>> f(0, 2, 50) Here is the square: 24 48 26 2 It is a good square. Ordering the elements from left to right column, from top to bottom, yields: 2 26 24 48 >>> f(0, 3, 100) Here is the square: 49 97 53 5 33 65 62 51 38 It is a good square. Ordering the elements from left to right column, from top to bottom, yields: 5 49 62 33 51 65 38 53 97 >>> f(0, 6, 5000) Here is the square: 3155 3445 331 2121 4188 3980 3317 2484 3904 2933 4779 1789 4134 1140 2308 1144 776 2052 4362 4930 1203 2540 809 604 2704 3867 4585 824 2898 3556 2590 1675 4526 3907 3626 4270 It is a good square. Ordering the elements from left to right column, from top to bottom, yields: 331 1144 2308 2933 3867 4270 604 1203 2484 3155 3904 4362 776 1675 2540 3317 3907 4526 809 1789 2590 3445 3980 4585 824 2052 2704 3556 4134 4779 1140 2121 2898 3626 4188 4930

Question 7

Write a function that accepts a strictly positive integer called height and displays a triangle shape of numbers starting from 0 and of height height. Use only digits from 0 to 9 to construct the shape as per the examples below: >>> f(1) 0 >>> f(2) 0 123 >>> f(3) 0 123 45678 >>> f(4) 0 123 45678 9012345 >>> f(5) 0 123 45678 9012345 678901234 >>> f(6) 0 123 45678 9012345 678901234 56789012345 >>> f(20) 0 123 45678 9012345 678901234 56789012345 6789012345678 901234567890123 45678901234567890 123456789012345678 012345678901234567890 123456789012345678901234 456789012345678901234567890 901234567890123456789012345 678901234567890123456789012345 567890123456789012345678901234567 678901234567890123456789012345678 901234567890123456789012345678901234 456789012345678901234567890123456789 123456789012345678901234567890123456789

Question 8

Write a function that accepts a string of DISTINCT UPPERCASE letters only called letters and displays all pairs of words using all (distinct) letters in letters.Please note that the words need to be valid. Use the provided dictionary.txt to check the validity of words. Here are some execution examples: >>> f('ABCDEFGH') There is no solution >>> f('GRIHWSNYP') The pairs of words using all (distinct) letters in "GRIHWSNYP" are: ('SPRING', 'WHY') >>> f('ONESIX') The pairs of words using all (distinct) letters in "ONESIX" are: ('ION', 'SEX') ('ONE', 'SIX') >>> f('UTAROFSMN') The pairs of words using all (distinct) letters in "UTAROFSMN" are: ('AFT', 'MOURNS') ('ANT', 'FORUMS') ('ANTS', 'FORUM') ('ARM', 'FOUNTS') ('ARMS', 'FOUNT') ('AUNT', 'FORMS') ('AUNTS', 'FORM') ('AUNTS', 'FROM') ('FAN', 'TUMORS') ('FANS', 'TUMOR') ('FAR', 'MOUNTS') ('FARM', 'SNOUT') ('FARMS', 'UNTO') ('FAST', 'MOURN') ('FAT', 'MOURNS') ('FATS', 'MOURN') ('FAUN', 'STORM') ('FAUN', 'STROM') ('FAUST', 'MORN') ('FAUST', 'NORM') ('FOAM', 'TURNS') ('FOAMS', 'RUNT') ('FOAMS', 'TURN') ('FORMAT', 'SUN') ('FORUM', 'STAN') ('FORUMS', 'NAT') ('FORUMS', 'TAN') ('FOUNT', 'MARS') ('FOUNT', 'RAMS') ('FOUNTS', 'RAM') ('FUR', 'MATSON') ('MASON', 'TURF') ('MOANS', 'TURF')

Python 3 Cheat Sheet

Latest version on :
https://perso.limsi.fr/pointal/python:memento

Base Types

integer, float, boolean, string, bytes

```
int 783  0 -192  0b010 0o642 0xF3
          zero    binary  octal  hexa
float 9.23 0.0  -1.7e-6
                    ×10^-6
bool True  False
str "One\nTwo"   escaped new line
'I\'m'   escaped '
"""X\tY\tZ
1\t2\t3"""   escaped tab
bytes b"toto\xfe\775"
        hexadecimal  octal
```

Multiline string:

immutables

Container Types

- **ordered sequences**, fast index access, repeatable values

```
list [1,5,9]   ["x",11,8.9]   ["mot"]   []
tuple (1,5,9)   11,"y",7.4   ("mot",)   ()
```

Non modifiable values (immutables) — *expression with only comas →* `tuple`

```
str bytes   (ordered sequences of chars / bytes)
""   b""
```

- **key containers**, no *a priori* order, fast key access, each key is unique

```
dictionary dict {"key":"value"}   dict(a=3,b=4,k="v")   {}
(key/value associations) {1:"one",3:"three",2:"two",3.14:"π"}
collection set {"key1","key2"}   {1,9,3,0}   set()
```

keys=hashable values (base types, immutables…) `frozenset` *immutable set* *empty*

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 for
```

Variables assignment

`=`

assignment ⇔ **binding** of a *name* with a *value*

1) evaluation of right side expression value
2) assignment in order with left side names

```
x=1.2+8+sin(y)
a=b=c=0   assignment to same value
y,z,r=9.2,-7.6,0   multiple assignments
a,b=b,a   values swap
a,*b=seq
*a,b=seq   unpacking of sequence in item and list
x+=3   increment ⇔ x=x+3
x-=2   decrement ⇔ x=x-2
x=None   « undefined » constant value
del x   remove name x
```

and `*=` `/=` `%=` …

Conversions

`type(`*expression*`)`

```
int("15") → 15
int("3f",16) → 63   can specify integer number base in 2nd parameter
int(15.56) → 15   truncate decimal part
float("-11.24e8") → -1124000000.0
round(15.56,1)→15.6   rounding to 1 decimal (0 decimal → integer number)
bool(x)  False for null x, empty container x , None or False x ; True for other x
str(x)→"…"   representation string of x for display (cf. formatting on the back)
chr(64)→'@'   ord('@')→64   code ↔ char
repr(x)→"…"   literal representation string of x
bytes([72,9,64]) → b'H\t@'
list("abc") → ['a','b','c']
dict([(3,"three"),(1,"one")]) → {1:'one',3:'three'}
set(["one","two"]) → {'one','two'}
separator str and sequence of str → assembled str
   ':'.join(['toto','12','pswd']) → 'toto:12:pswd'
str splitted on whitespaces → list of str
   "words with  spaces".split() → ['words','with','spaces']
str splitted on separator str → list of str
   "1,4,8,2".split(",") → ['1','4','8','2']
sequence of one type → list of another type (via list comprehension)
   [int(x) for x in ('1','29','-3')] → [1,29,-3]
```

Sequence Containers Indexing

for lists, tuples, strings, bytes…

```
negative index   -5  -4  -3  -2  -1
positive index    0   1   2   3   4
lst=[10, 20, 30, 40, 50]
positive slice    0   1   2   3   4   5
negative slice   -5  -4  -3  -2  -1
```

Items count

`len(lst)→5`

index from 0
(here from 0 to 4)

Individual access to **items** via `lst[`*index*`]`

```
lst[0]→10   ⇒ first one   lst[1]→20
lst[-1]→50   ⇒ last one   lst[-2]→40
```

On mutable sequences (`list`), remove with `del lst[3]` *and modify with assignment* `lst[4]=25`

Access to **sub-sequences** via `lst[`*start slice*`:`*end slice*`:`*step*`]`

```
lst[:-1]→[10,20,30,40]   lst[::-1]→[50,40,30,20,10]   lst[1:3]→[20,30]   lst[:3]→[10,20,30]
lst[1:-1]→[20,30,40]   lst[::-2]→[50,30,10]   lst[-3:-1]→[30,40]   lst[3:]→[40,50]
lst[::2]→[10,30,50]   lst[:]→[10,20,30,40,50]   shallow copy of sequence
```

Missing slice indication → from start / up to end.
On mutable sequences (`list`), remove with `del lst[3:5]` *and modify with assignment* `lst[1:4]=[15,25]`

Boolean Logic

Comparisons : `<` `>` `<=` `>=` `==` `!=` (≤ ≥ = ≠)
(boolean results)

`a and b` logical and — *both simulta-neously*

`a or b` logical or — *one or other or both*

pitfall : `and` *and* `or` *return* *value* *of* `a` *or of* `b` *(under shortcut evaluation).*
⇒ *ensure that* `a` *and* `b` *are booleans.*

`not a` logical not

`True` `False` — True and False constants

Statements Blocks

```
parent statement:
    statement block 1…
    ⋮
parent statement:
    statement block2…
    ⋮
next statement after block 1
```

indentation !

configure editor to insert 4 spaces in place of an indentation tab.

Modules/Names Imports

module `truc` ⇔ *file* `truc.py`

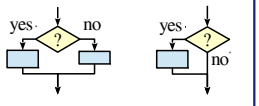
```
from monmod import nom1,nom2 as fct
   →direct access to names, renaming with as
import monmod   →access via monmod.nom1 …
```

modules and packages searched in *python path* (cf `sys.path`)

Conditional Statement

statement block executed only if a condition is true

`if` *logical condition*`:`
→ *statements block*



Can go with several *elif*, *elif*... and only one final *else*. Only the block of first true condition is executed.

```
if age<=18:
    state="Kid"
elif age>65:
    state="Retired"
else:
    state="Active"
```

with a var `x`*:*
`if bool(x)==True:` ⇔ `if x:`
`if bool(x)==False:` ⇔ `if not x:`

Maths

floating numbers… approximated values

Operators: `+` `-` `*` `/` `//` `%` `**`
Priority `(…)`
× ÷ ↑ ↑ a^b
integer ÷ ÷ remainder

`@` → matrix × *python3.5+*`numpy`

```
(1+5.3)*2→12.6
abs(-3.2)→3.2
round(3.57,1)→3.6
pow(4,3)→64.0
```

usual order of operations

angles in radians

```
from math import sin,pi…
sin(pi/4)→0.707…
cos(2*pi/3)→-0.4999…
sqrt(81)→9.0   √
log(e**2)→2.0
ceil(12.5)→13
floor(12.5)→12
```

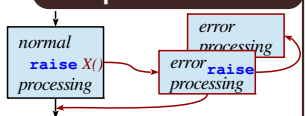
modules `math`, `statistics`, `random`, `decimal`, `fractions`, `numpy`, *etc. (cf. doc)*

Exceptions on Errors

Signaling an error:
`raise` *ExcClass(…)*

Errors processing:

```
try:
    →normal procesising block
except Exception as e:
    →error processing block
```



`finally` *block for final processing in all cases.*

Conditional Loop Statement

*statements block executed **as long as** condition is true*

```
while logical condition:
    statements block
```


```
s = 0          # initializations before the loop
i = 1          # condition with a least one variable value (here i)
while i <= 100:
    s = s + i**2
    i = i + 1  # make condition variable change !
print("sum:",s)
```

Algo:

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

beware of infinite loops!

Loop Control

`break` *immediate exit*
`continue` *next iteration*
`else` *block for **normal** loop exit.*

Display

```
print("v=",3,"cm :",x,",",y+4)
```

items to display : literal values, variables, expressions

`print` options:

- `sep=" "` items separator, default space
- `end="\n"` end of print, default new line
- `file=sys.stdout` print to file, default standard output

Input

```
s = input("Instructions:")
```

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

Generic Operations on Containers

`len(c)` → items count
`min(c)` `max(c)` `sum(c)`
`sorted(c)` → `list` sorted *copy*
`val in c` → boolean, membership operator `in` (absence `not in`)
`enumerate(c)` → *iterator* on (index, value)
`zip(c1,c2…)` → *iterator* on tuples containing c_i items at same index
`all(c)` → `True` if **all** `c` items evaluated to true, else `False`
`any(c)` → `True` if **at least one** item of `c` evaluated true, else `False`

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

*Specific to **ordered sequences containers** (lists, tuples, strings, bytes…)*

`reversed(c)` → *inversed iterator* `c*5`→ duplicate `c+c2`→ concatenate
`c.index(val)` → *position* `c.count(val)` → *events count*

```
import copy
copy.copy(c) → shallow copy of container
copy.deepcopy(c) → deep copy of container
```

Operations on Lists

modify original list

`lst.append(val)`	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 *val*
`lst.pop([idx])` →*value*	remove & return item at index *idx* (default last)
`lst.sort()` `lst.reverse()`	sort / reverse liste *in place*

Operations on Dictionaries

`d[key]=value` `d.clear()`
`d[key]` → *value* `del d[key]`
`d.update(d2)` update/add associations
`d.keys()` `d.values()` `d.items()` →*iterable views on keys/values/associations*
`d.pop(key[,default])` → *value*
`d.popitem()` → *(key,value)*
`d.get(key[,default])` → *value*
`d.setdefault(key[,default])` →*value*

Operations on Sets

Operators:
`|` → union (vertical bar char)
`&` → intersection
`-` `^` → difference/symmetric diff.
`<` `<=` `>` `>=` → inclusion relations
Operators also exist as methods.

`s.update(s2)` `s.copy()`
`s.add(key)` `s.remove(key)`
`s.discard(key)` `s.clear()`
`s.pop()`

Files

storing data on disk, and reading it back

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

- file **variable** for operations
- **name** of file on disk (+path…) — cf. modules `os`, `os.path` and `pathlib`
- opening **mode**: `'r'` read, `'w'` write, `'a'` append, …`'+'` `'x'` `'b'` `'t'`
- **encoding** of chars for *text files*: `utf8` `ascii` `latin1` …

writing

`f.write("coucou")`
`f.writelines(list of lines)`

reading

read empty string if end of file
`f.read([n])` → *next chars* — *if n not specified, read up to end !*
`f.readlines([n])` → `list` *of next lines*
`f.readline()` → *next line*

*text mode `t` by default (read/write `str`), possible binary mode `b` (read/write `bytes`). **Convert from/to required type** !*

`f.close()` dont forget to **close the file** after use !

`f.flush()` write cache
`f.truncate([size])` resize

reading/writing progress sequentially in the file, modifiable with:
`f.tell()` →*position*
`f.seek(position[,origin])`

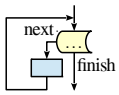
Very common: opening with a guarded block (automatic closing) and reading loop on lines of a text file:

```
with open(…) as f:
    for line in f :
        # processing of line
```

Iterative Loop Statement

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

```
for var in sequence:
    statements block
```



Go over sequence's **values**

```
s = "Some text"   # initializations before the loop
cnt = 0
for c in s:       # loop variable, assignment managed by for statement
    if c == "e":
        cnt = cnt + 1
print("found",cnt,"'e'")
```

Algo: count number of `e` in the string.

loop on dict/set ⇔ loop on keys sequences
use *slices* to loop on a subset of a 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)
```

Algo: limit values greater than 15, memorizing of lost values.

Go simultaneously over sequence's **index** and **values**:
`for idx,val in enumerate(lst):`

good habit : don't modify loop variable

Integer Sequences

`range([start,] end [,step])`

start default 0, *end* not included in sequence, *step* signed, default 1

`range(5)` → 0 1 2 3 4 `range(2,12,3)` → 2 5 8 11
`range(3,8)` → 3 4 5 6 7 `range(20,5,-5)` → 20 15 10
`range(len(seq))` → *sequence of index of values in seq*

range provides an immutable sequence of int constructed as needed

Function Definition

function name (identifier)
named parameters

```
def fct(x,y,z):
    """documentation"""
    # statements block, res computation, etc.
    return res
```

`fct`

result value of the call, if no computed result to return: `return None`

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

Advanced: `def fct(x,y,z,*args,a=3,b=5,**kwargs):`
**args variable positional arguments (→`tuple`), default values,*
***kwargs variable named arguments (→`dict`)*

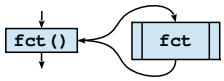
Function Call

```
r = fct(3,i+2,2*i)
```

storage/use of returned value — *one argument per parameter*

this is the use of function name *with parentheses* which does the call

*Advanced: *sequence **dict*



Operations on Strings

`s.startswith(prefix[,start[,end]])` `s.endswith(suffix[,start[,end]])` `s.strip([chars])`
`s.count(sub[,start[,end]])` `s.partition(sep)` → *(before,sep,after)*
`s.index(sub[,start[,end]])` `s.find(sub[,start[,end]])`
`s.is…()` *tests on chars categories (ex. `s.isalpha()`)*
`s.upper()` `s.lower()` `s.title()` `s.swapcase()`
`s.casefold()` `s.capitalize()` `s.center([width,fill])`
`s.ljust([width,fill])` `s.rjust([width,fill])` `s.zfill([width])`
`s.encode(encoding)` `s.split([sep])` `s.join(seq)`

Formatting

formating directives — values to format

```
"modele{} {} {}".format(x,y,r) → str
```

`"{`*selection* `:` *formatting* `!` *conversion*`}"`

- **Selection** :
  `2`
  `nom`
  `0.nom`
  `4[key]`
  `0[2]`

Examples:

```
"{:+2.3f}".format(45.72793)
→'+45.728'
"{1:>10s}".format(8,"toto")
→'      toto'
"{x!r}".format(x="I'm")
→'"I\'m"'
```

- **Formatting** :

fill char *alignment* *sign* *mini width*.*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),
string: `s` … `%` percent

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