CS145 Fall 2023 – Midterm

Name: ______________________________ **Section: A / B**

Student ID Number: ______________________________

Date: ________________ **Start time:** ____________ **End time:** ____________

Honor Code:

Signature: ________________________

This exam is closed book, closed notes, closed computer, closed calculator, etc. You may only use (1) the midterm "cheat sheet" provided with this exam and (2) a single double-sided letter sheet of notes of your own creation. **You have 2 hours.** Read the problem descriptions carefully and write your answers **clearly, legibly, in the space provided**. Circle or otherwise indicate your answer if it might not be easily identified. You may use extra sheets of paper, stapled to your exam, if you need more room, as long as the problem number is clearly labeled and your name is on the paper. If you attached extra sheets indicate in the provided space for the problem to look for the extra sheets for that problem.
You do need to include module imports (if relevant for your code), but do not need to include comments or docstrings in your code.

Question	Points	Score
Loop Warm-Up	20	
True or False	30	
Function Calls	15	
Email Validation	45	
Debugging, Testing, and Documentation	25	
Shapes and Turtles	15	
Total:	150	

Question 1: Loop Warm-Up [20 points]

Write two loops that print every other character in a string `s`, starting with the character at index `0`. For example, if `s = "otter"`, the output should be:

```
o
t
r
```

One should be a **for** loop using `range`, and the other should be a `while` loop.

(a) `for` loop

(b) `while` loop

Question 2: True or False [30 points]

For each of the statements below state whether they are **T** (true) or **F** (false).

(a) ________ The expression `(5 + 7) % 2 == 0 or x` always evaluates to `True`, regardless of the value of `x` (as long as x is defined)

(b) ________ All recursive functions must include a `return` statement

(c) ________ The binary search algorithm makes the assumption that the input list is sorted

(d) ________ `for` loops can always do the same thing as `while` loops with less code

(e) ________ The following loop is infinite

```
count = 0
while count != 10:
    print("Does it stop?")
    count += 3
```

(f) ________ If there is an `elif` in your function, there must be an `else`

(g) ________ The base case of a recursive function should always return an empty string if the input to the function is a string

(h) ________ There is an input to the function below that will cause it to print both `"bananas"` and `"oranges"`

```
def mystery(arg):
    if arg > 20:
        print("bananas")
    elif arg > 10:
        print("oranges")
    else:
        print("grapefruit")
```

(i) ________ If you ran this code in Thonny, `20` would be printed

```
x = 20
if x % 5 == 0:
    print(x)
else:
    print("nope")
x = y
```

(j) ________ y and z have the same value after executing this code:

```
def mystery(arg):
    for x in arg:
        if x == "c":
            return True
    return False

z = mystery("according")
y = mystery(["c", "a", "t"])
```

NOTE: Due to a syntax error in this question (now fixed), everyone got credit.

Question 3: Function Calls [15 points]

Consider the following Python code:

```
def bar(x, z):
    if z > x:
        return z
    return 0

def foo(l):
    y = 0
    for i in range(len(l)):
        y += bar(i, l[i])
    return y

z = foo([4, 1, 3, 9])
```

After execution the value of z is:

3. ____________

For partial credit (in case your answer above is incorrect), fill out the table below with the calls that will be made to the bar function in order, and the values that are returned. The first function call is given to you as an example. There may be extra spaces in the table that you do not need.

Order	**Function Call**	**Returns**
1st call to `bar`	`bar(0, 4)`	4
2nd call to `bar`		
3rd call to `bar`		
4th call to `bar`		
5th call to `bar`		
6th call to `bar`		
7th call to `bar`		
8th call to `bar`		

Question 4: Email Validation [45 points]

For this question, you will write functions that help you to get a valid Middlebury email address from a user.

(a) (15 points) `endswith` function

Write a function that determines whether a string `s1` **ends with** another string `s2`. You may assume that `s1` is at least as long as `s2`. Here are some example function calls and outputs:

Function Call	**Returns**
`endswith("username@middlebury.edu", "@middlebury.edu")`	`True`
`endswith("username@gmail.com", "@middlebury.edu")`	`False`
`endswith("horse", "e")`	`True`

HINT: this function does not require a loop, and can be written succinctly using string slicing, boolean operators, and built-in functions. You **may not** use the string `endswith` method (which we have not discussed in class).

Full credit will be given to concise answers written using two lines.

NOTE: Answers that only work if `s2` was not an empty string were also accepted.

(b) (15 points) `has_spaces` function

Next, write a **recursive** `has_spaces` function that determines a string has any spaces in it.

Here are some example function calls and outputs:

Function Call	**Returns**
`has_spaces("tswift")`	`False`
`has_spaces("paul mccartney")`	`True`
`has_spaces("beyonce")`	`False`

You can use the `is_space` function written below to determine whether or not a character is a space (spaces and empty strings may look similar when handwritten).

```
def is_space(char):
    return char == " "
```

(c) (15 points) `get_email` function

Finally, we'll put the pieces together to write a function that gets input from a user and returns it only after checking that they have written a valid middlebury email address. If the email is not valid, it will ask for their email again. For the sake of this problem, a valid email address must end with `@middlebury.edu` and must have no spaces (that means that simply `"@middlebury.edu"` is technically valid). Write the function by **re-organizing** the lines that are provided here (you must use all lines). **Use the table below**: input the line number in the first column to order the lines, then indicate the indentation level by writing at least the first two characters of each line using the grey lines as guidelines for the size of a tab. Any correct solution will be accepted.

NOTE: the use of `while True` here means that we will keep executing the body of the loop until we return a value.

```
1  return user_email
2  def get_email():
3  user_email = input("Email: ")
4  domain = "@middlebury.edu"
5  if endswith(user_email, domain) and not has_spaces(user_email):
6  while True:
```

Line #	**Line text at correct indentation level (at least first 2 characters)**

Question 5: Debugging, Testing, and Documentation [25 points]

The questions on debugging, testing, and documentation will refer to the following function. The function **should return the largest number in a list with at least one number in it**, but it has three errors in it.

```
1 def largest_number(numbers):
2     curr_largest = 0
3     for num in numbers
4         if num > curr_largest:
5             curr_largest = number
6     return curr_largest
```

(a) There are 3 problems with this code, including: i) one syntax error, ii) one runtime error (syntactically valid Python that generates an error when actually executed) and iii) one logic error (the code would execute to completion if the other errors are fixed but produces incorrect results). For this question, you will identify and describe all three errors in the code. The errors should not be variations of the same issue and should impact correctness, not just style. You do not need to fix the errors.

i. (5 points) Syntax Error
Write the **line number** of the syntax error on the line:

i. ________________

Write a description of the syntax error here:

ii. (5 points) Runtime Error
Write the **line number** of the runtime error on the line:

ii. ________________

Write a description of the runtime error here:

iii. (5 points) Logic Error
Fill in the function call below as if you are using it to test the function. **Your function call must reveal the logic error in the code.** In other words, the returned value should be incorrect for this test case.

```
largest_number(                        )
```

Write a description of the logic error here:

(b) (3 points) We talked about three basic patterns when dealing with lists: map, reduce, and filter. Which type of function is this? Select one answer only, and **assume that all of the errors you identified have been fixed**.

- ◯ map
- ◯ filter
- ◯ reduce

(c) (7 points) Write an appropriate docstring for the function. **Assume that all of the errors you identified have been fixed**. Your docstring should include:

- A short sentence describing what the function does in simple terms.
- A description of the data type(s) that the argument `numbers` should have in order for the function to work.
- A description of the return value of the function, including its data type.

Question 6: Shapes and Turtles [15 points]

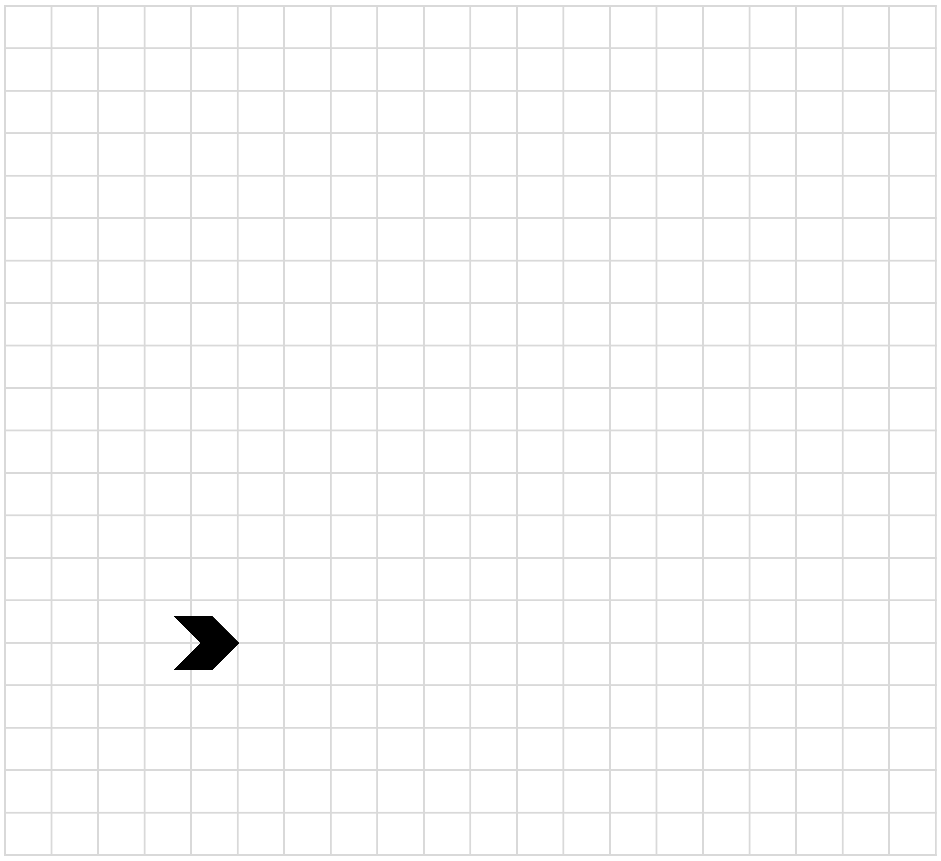
We used the `turtle` module in Lab 4 to draw pictures in Python. Draw the picture that is created by running this code:

```
import turtle

def f(length_pixels):
    if length_pixels > 1:
        for i in range(4):
            turtle.forward(length_pixels)
            turtle.left(90)
        f(length_pixels - 2)

f(10)
```

Assume that the turtle starts at the spot shown in the grid provided below, facing right, at the rightmost tip of the shape. Treat every square on the grid as 1 pixel.



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"          Multiline string:
      escaped new line      """X\tY\tZ
       'I\'m'               1\t2\t3"""
       escaped '            escaped tab
bytes b"toto\xfe\775"
          hexadecimal octal
```

☝ *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`	*unpacking of sequence in*
`*a,b=seq`	*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`

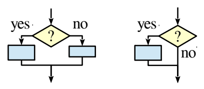
`import monmod` →*access via* `monmod.nom1` …

☝ 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* *angles in radians*

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

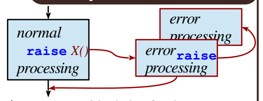
`@` → matrix × *python3.5+*`numpy`

```
(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*

```
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
```



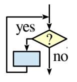
☝ `finally` *block for final processing in all cases.*

Conditional Loop Statement

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

beware of infinite loops!

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



```
s = 0
i = 1
while i <= 100:
    s = s + i**2
    i = i + 1
print("sum:",s)
```

*initializations **before** the loop*

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

make condition variable change !

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

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)`

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

`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`

*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)
copy.deepcopy(c)
```

`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…)

opening **mode**
- `'r'` read
- `'w'` write
- `'a'` append
- …`'+'` `'x'` `'b'` `'t'`

encoding of chars for *text files*: `utf8` `ascii` `latin1` …

cf. modules `os`, `os.path` and `pathlib`

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]*`)`

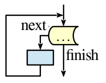
Very common: opening with a guarded block (automatic closing) and reading loop on lines of a text file:

```
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"
cnt = 0
for c in s:
    if c == "e":
        cnt = cnt + 1
print("found",cnt,"'e'")
```

*initializations **before** the loop*

*loop variable, **assignment managed by `for` statement***

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

good habit : don't modify loop variable

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):
```

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*

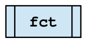
range provides an immutable sequence of int constructed as needed

Function Definition

function name (identifier)

named parameters

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



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

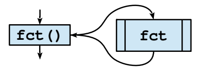
```
r = fct(3,i+2,2*i)
```

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** :

<u>*fill char*</u> <u>*alignment*</u> <u>*sign*</u> <u>*mini width*</u> . <u>*precision~maxwidth*</u> <u>*type*</u>

`<` `>` `^` `=` `+` `-` *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)