# Tentamen i Programmering DI2006, Del 1

**Datum**: 2024-04-02
**Tid**: 9.00–12.00
**Ansvarig lärare**: Eric Järpe (tel: 0729-77 36 26, email: eric.jarpe@hh.se)

---

Anvisningar

- Tillåtna hjälpmedel är
  - formelsamling (som är häftad till tentamenstexten)
  - miniräknare TI-30Xa (Texas Instruments)
  - skrivpapper
  - penna
  - suddigummi
  - linjal
  - frukt, fika
- Till varje uppgift finns angivet hur många poäng som maximalt utdelas för uppgiften.
- Tentamen består av två delar: **Del 1** och **Del 2**.
- Samtliga frågor i Del 1 ska besvaras i den svarstalong som är bifogad med tentamenstexten.
- Del 2 ges på fredag.
- Då programkod anges som svar ska den vara i så körbart skick som möjligt.
- Del 1 består av 22 frågor och här kan man maximalt få 30 poäng.
- För betyg 3 från Del 1 måste man ha minst 15 poäng.
- Resultatet från Del 1 kan bara ge godkänt med betyg 3 oavsett hur många poäng man har.
- För högre betyg måste man först ha godkänt på Del 1 och sedan skriva tillräckligt många poäng på Del 2.

*LYCKA TILL!*

# Del 1

## Flervalsfrågor

1. Om en variabel `x` tilldelas värdet 50.0 så blir variabelns typ: (1p)
   (a) `int`
   (b) `num`
   (c) `char`
   (d) `bool`
   (e) `float`
   (f) `str`
   (g) Inget av de ovanstående alternativen

---

2. Räknesättet som returnerar resten vid heltalsdivision mellan två tal är: (1p)
   (a) `*`
   (b) `**`
   (c) `%`
   (d) `%%`
   (e) `/`
   (f) `//`
   (g) Inget av de ovanstående alternativen

---

3. För blanda en virtuell kortlek genereras först kortleken:

```
farger = ['Hj', 'Sp', 'Ru', 'Kl']
valorer = ['Ess', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'Knekt',
    'Dam', 'Kung']
kortlek = []
for i in range(4):
    for j in range(13):
        kortlek.append(farger[i]+' '+valorer[j])
```

och blandas sedan:

```
import random
blandad_kortlek = []
for i in range(52):
    blandad_kortlek.append(random.choice(kortlek))
```

Men detta blir inte bra, för vissa kort kommer med flera gånger och andra kort kommer inte med alls. Korrigera detta så att alla kort kommer med blandade, en och endast en gång genom att byta ut den sista raden mot: (1p)

   (a) `blandad_kortlek.append(kortlek.pop(random.choice(range(52-i))))`
   (b) `blandad_kortlek = blandad_kortlek + [random.choice(kortlek)]`
   (c) `blandad_kortlek.pop(random.choice(kortlek))`

(d) `blandad_kortlek = kortlek[random.randint(1,len(kortlek-1))]`

(e) `blandad_kortlek = blandad_kortlek + [random.randint(0,len(kortlek)-1)]`

(f) `blandad_kortlek.append(kortlek[random.randint(0,len(kortlek)-1)])`

(g) Inget av de ovanstående alternativen

---

4. Hur skriver man för att ange square root of 2 med 5 decimalers noggrannhet? (1p)

(a) `format(print(2**0.5),'.5f')`

(b) `input(print(2**0.5),'.5f')`

(c) `print(sqrt(2),format('.5f'))`

(d) `print(format(5**1/2),'.2f')`

(e) `format(print(5**0.2,'.2f'))`

(f) `print(format(2**0.5,'.5f'))`

(g) Inget av de ovanstående alternativen

---

5. Vilket av följande uttryck kan inte vara booleskt? (1p)

(a) `5>t`

(b) `3.0!=y/0.1`

(c) `5 in range(G)`

(d) `x%4=3.14`

(e) `8//3==9/2`

(f) `z**z+4>=1e5`

(g) Inget av de ovanstående alternativen

---

6. Vilket kommando kan förutom `if` och `elif` kan användas för att stapla en sekvens av booleska uttryck på varandra? (1p)

(a) `for`

(b) `else`

(c) `while`

(d) `return`

(e) `next`

(f) `def`

(g) Inget av de ovanstående alternativen

---

7. Talet 113 skrivet binärt som en byte blir: (1p)

(a) `10010110`

(b) `01110001`

(c) `00011011`

(d) `00001101`

(e) `A083CF19`

(f) `00000113`

(g) Inget av de ovanstående alternativen

---

8. Vilket av följande alternativ är *inte* ett korrekt variabelnamn? (1p)
   (a) `agaton`
   (b) `klockan_sex`
   (c) `Korrekt_Lösning`
   (d) `Antal_varv`
   (e) `kl_6_till_kl_9`
   (f) `i9elk0tt`
   (g) Inget av de ovanstående alternativen

---

9. Datatypen dictionary utgörs av element som är: (1p)
   (a) `kärnor`
   (b) `strukturer`
   (c) `par bestående av (key, value)`
   (d) `par bestående av (record, content)`
   (e) `tripplar bestående av (track, cluster, sector)`
   (f) `tripplar bestående av (x, y, z)`
   (g) Inget av de ovanstående alternativen

---

10. I Python exekveras vanligtvis koden genom att den skickas till en: (1p)
    (a) `server`
    (b) `kvantdator`
    (c) `preprocessor`
    (d) `kondensator`
    (e) `interpretator`
    (f) `radiator`
    (g) Inget av de ovanstående alternativen

---

11. För att spara tennisresultat i filen `resultat.txt` skrivs följande kod:

```
f = _____________________________
s1,s2 = input('Ange spelare 1: '), input('Ange spelare 2: ')

_____________________________
r1,r2,r3 = input('Set 1: '),input('Set 2: '),input('Set 3. Om ej, "N": ')
_____________________________
_____________________________
_____________________________
_____________________________
_____________________________
f.close()
```

Resultat från varje match ska presenteras på en rad och resultaten ska läggas till tidigare matcher som redan registrerats. Hur ska koden fullbordas? (1p)

(a)
```
Rad 1: open('resultat.txt','rt')
Rad 3: r0 = f.readline()
Rad 5: r1 = f.readline()
Rad 6: r2 = f.readline()
Rad 7: while r!='N':
Rad 8:   r3 = f.readline()
Rad 9:   r = 'N'
```

(b)
```
Rad 1: open('resultat.txt','at')
Rad 3: f.write(s1+' vs '+s2+': ')
Rad 5: f.write(r1+', ')
Rad 6: f.write(r2)
Rad 7: if r3!='N':
Rad 8:   f.write(', '+r3)
Rad 9: f.write('\n')
```

(c)
```
Rad 1: open('resultat.txt','r+')
Rad 3: s = f.readline()
Rad 5: s = f.readline()
Rad 6: s = f.readline()
Rad 7: while s!=r3:
Rad 8:   r3 = f.readline()
Rad 9: f.write('\t')
```

(d)
```
Rad 1: open('resultat.txt','rb')
Rad 3: r0 = f.readline()
Rad 5: r1 = f.readline()
Rad 6: r2 = f.readline()
Rad 7: while r0!=r1:
Rad 8:   f.read()
Rad 9:   r0 = r1
```

(e)
```
Rad 1: open('resultat.txt','xt')
Rad 3: f.write(s1+s2)
Rad 5: f.write(r1)
Rad 6: f.write(r2)
Rad 7: while(s1==s2):
Rad 8:   f.write(r3+s1+s2)
Rad 9:   s1 = s2+'\n'
```

(f)
```
Rad 1: open('resultat.txt','wt')
Rad 3: f.write(s1+'\t'+s2+'\n')
Rad 5: f.write(r1+'\t'+r2+'\n')
Rad 6: f.write(r2+'\t'+r3+'\n')
Rad 7: if s1!=s2:
Rad 8:   s2=s1
Rad 9:   f.write(r+'\t'+r+\n')
```

(g) Inget av de ovanstående alternativen

---

12. Följande rekursiva kod hittas:

```
1. def f(a):
2.   if len(a)>8:
3.     return a[0]+a[2:4]
4.   elif len(a)>3:
5.     return a[1:]
6.   else:
7.     return a[0]+f(a+a[0])
8.
9. a = input('Ange en sträng: ')
10. print(f(a))
```

Vid exekvering av koden ger indata `'Högskolan'` utdata `'Hgs'`. Ange i vilken ordning programraderna 1–10 utförs. (1p)

(a) `1, 2, 3, 4, 5, 6, 7, 8, 9, 10`

(b) `1, 2, 4, 6, 7, 1, 7`

(c) `9, 10, 3`

(d) `9, 10, 1, 2, 3`

(e) `1, 7, 10`

(f) `9, 10, 1, 3, 4, 5, 6, 7`

(g) Inget av de ovanstående alternativen

---

13. Den metod som används för att skapa ett `object` är: (1p)

(a) `__object__`

(b) `__str__`

(c) `object`

(d) `init`

(e) `__init__`

(f) `__def__`

(g) Inget av de ovanstående alternativen

---

14. Hur kan man på en rad tilldela variablen `a` värdet 5 (av typen `int`) och variabeln `b` värdet `True` (av typen `bool`)? (1p)

(a) `a = 5, b = True`

(b) `t = (5, True); a = t[0]; b = t.pop(1)`

(c) `a, b = 10//2, True or False`

(d) `a = 5.0 and b = True`

(e) `a = 25/5 + b = 1==1.0`

(f) `t = [not(1==1.0), 5e0]; t.reverse(); [a, b] = t`

(g) Inget av de ovanstående alternativen

---

## Skrivfrågor

15. Vad är den 3-bokstaviga förkortningen för de datorkomponent som man brukar kalla "datorns hjärna"? (1p)

---

16. Betrakta koden:

```
a,b = 0,1
if a*b:
  c = int(str(a)+str(b))
else:
  c = a**a/b
```

Vilken typ får variabeln `c`? (1p)

---

17. Hur kan man på en rad kolla om listan `lista` innehåller några dubletter? (2p)

---

18. Hur kan man på en rad skapa en lista med de udda talen från och med 33 till och med 77? (2p)

---

19. Skriv en funktion, `revnum`, på två rader som givet ett tal, `x`, returnerar det tal som består av samma siffror som fast i omvänd ordning. Till exempel ska `revnum(123)` resultera i svaret `321`. (2p)

---

20. Vad skrivs ut om man exekverar koden:

```
a = 'påskharen'
while a[5]!='n':
  a = a[2:]+a[:2]

print(a)
```

? (2p)

---

21. Hur kan man skriva kod som givet tupeln

```
c = ('Rött', 'Grönt', 'Gult', 'Blått', 'Svart', 'Vitt')
```

gör att `c` får värdet av 4 slumpmässigt valda färger? (3p)

---

22. Vid inloggning till ett datasystem anges ett lösenord som kontrolleras mot en lista som finns sparad i systemet. För att göra det säkrare mot intrång sparas inte någon lista av de egentliga lösenorden utan bara ett hashvärde av lösenordet, dvs ett tal som kan beräknas givet att man vet lösenordet men lösenordet är svårt att gissa om man bara har talet. Programmet gatekeeper.py som är ett enkelt program för att göra inloggningen består av koden:

```
def hash(passw):
  n = int(''.join([str(ord(c)) for c in list(passw)]))
  return (n*(n+3))%882377

system_passwords = {'arjo':418651, 'beru':461374, 'agol':39175, 'stan':790774}
u = input('User: ')
p = hash(input('Password: '))
fel = True
j=0
while(fel and j<2):
  if u not in system_passwords.keys():
    print('Fel användarnamn! Försök igen.')
    u = hash(input('User: '))
    j=j+1
  elif system_passwords[u]!=p:
    print('Fel lösenord! Försök igen.')
    p = input('Password: ')
    j=j+1
  if u in system_passwords.keys() or system_passwords[u]==p:
    print('Välkommen att använda systemet!')
    fel = False
if fel:
  print('Denna incident har rapporterats till systemadministratören.')
```

Det fungerar dock inte riktigt som det ska – ibland släpper den in användare trots att de angivit felaktiga inloggningsuppgifter och ibland släpper det inte in en som angivit korrekta uppgifter på andra eller tredje försöket. Hur ska man korrigera koden på tre rader så att det fungerar tillfredsställande? (3p)

---

# 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`
☹ ~~`for`~~ ~~`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      and *= /= %= …
x-=2   decrement ⇔ x=x-2
x=None « undefined » constant value
del x  remove name x
```

## 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 simultaneously*

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

pitfall : `and` *and* `or` *return* *value* *of* `a` *or* `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`

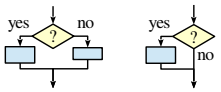
```
import monmod
```
→*access via* `monmod.nom1` …

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 `(…)` — × ÷ ↑ ↑ $a^b$ — 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*

```
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
```

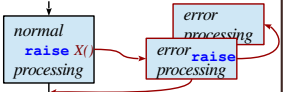
modules `math`, `statistics`, `random`, `decimal`, `fractions`, `numpy`, *etc. (cf. doc)*

## 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          }
while i <= 100:   # condition with a least one variable value (here i)
    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*

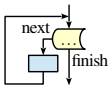
`continue` *next iteration*

`else` *block for **normal** loop exit.*

## 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

## 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
```

## 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*

## 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()`

## 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`)*

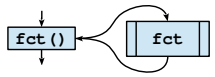
## Function Call

```
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)

## 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
```