## Conditional Loop Statement

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

```
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 !

beware of infinite loops!

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

### Loop Control

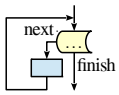
| | |
|---|---|
| `break` | immediate exit |
| `continue` | next iteration |

`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"
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

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.*

good habit : don't modify loop variable

Go simultaneously over sequence's **index** and **values**:

```
for idx,val in enumerate(lst):
```

## 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* |

## 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`)*

## 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 Call

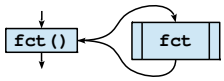
```
r = fct(3,i+2,2*i)
```

*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)`

## 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
```

## 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)