

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


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

↑
immutable
as an ordered sequence of chars
- no *a priori* order, unique key, fast key access ; keys = base types or tuples


```
dict {"key": "value"} {}
      {1: "one", 3: "three", 2: "two", 3.14: "pi"}
```

dictionary
key/value associations
- ```
set {"key1", "key2"} {1,9,3,0} set()
```

## 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)
```

↑  
value or computed expression  
variable name (identifier)

```
y,z,r = 9.2,-7.6,"bad"
```

variables names  
container with several values (here a tuple)

**x+=3** ← increment  
decrement → **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 formatting allowing finer control
```

**bool** → use comparators (with ==, !=, <, >, ...), logical boolean result

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

**":".join(["toto","12","pswd"])** → **"toto:12:pswd"**  
joining string sequence of strings

**"words with spaces".split()** → **['words','with','spaces']**  
splitting string

**"1,4,8,2".split(",")** → **['1','4','8','2']**

## 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<sup>b</sup>  
integer ÷ ÷ remainder

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

```
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)
```

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 + i\*\*2**

**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"** } initializations before the loop

**cnt = 0**

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 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

**d.keys()** } associations

**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 bloc only exist in the block and during the function call ("black box")

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

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

**Function call**

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

formatting directives

**Strings formatting**

values to format

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

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

□ **Selection:**

**2**

**x**

**0.nom**

**4[key]**

**0[2]**

□ **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)