

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
bool True False
str "One\nTwo"
  escaped new line
  'I\'m'
  escaped '
bytes b"toto\xfe\775"
      hexadecimal octal
  
```

Multiline string: `"""X\tY\tZ\n1\t2\t3"""`
escaped tab

☞ **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",)
- key containers**, no a priori order, fast key access, each key is unique
 - dictionary dict** {"key": "value"} dict (a=3, b=4, k="v")
 - collection set** {"key1", "key2"} {1,9,3,0}
 - frozenset** immutable set
 - set** {}
 - empty set** set ()

☞ **Non modifiable values (immutables)** ☞ expression with only commas → tuple
☞ **keys=hashable values (base types, immutables...)**

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**
☞ **9y and for**

Conversions

type (expression)

```

int ("15") → 15
int ("3f", 16) → 63
int (15.56) → 15
float ("-11.24e8") → -112400000.0
round(15.56, 1) → 15.6
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
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]
  
```

can specify integer number base in 2nd parameter
truncate decimal part
rounding to 1 decimal (0 decimal → integer number)

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
y, z, r=9.2, -7.6, 0
a, b=b, a
a, *b=seq
*a, b=seq
x+=3
x-=2
x=None
del x
  
```

assignment to same value
multiple assignments
values swap
unpacking of sequence in item and list

☞ **increment ⇔ x=x+3**
☞ **decrement ⇔ x=x-2**
☞ **« undefined » constant value**
☞ **remove name x**

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
lst[-1] → 50
lst[1] → 20
lst[-2] → 40
  
```

☞ **first one** ☞ **last one**

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: -1] → [20, 30, 40]
lst[: : 2] → [10, 30, 50]
lst[ : -1] → [50, 40, 30, 20, 10]
lst[ : -2] → [50, 30, 10]
lst[ : ] → [10, 20, 30, 40, 50]
lst[1: 3] → [20, 30]
lst[-3: -1] → [30, 40]
lst[: 3] → [10, 20, 30]
lst[3: ] → [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 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...
│
│
└ statement block 2...
  
```

☞ **indentation !**
☞ **next statement after block 1**

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

Modules/Namespace Imports

module **truc** ⇔ file **truc.py**

```

from monmod import nom1, nom2 as fct
import monmod
  
```

☞ **direct access to names, renaming with as**
☞ **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 (...)
☞ 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 processing 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

Loop Control

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

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

beware of infinite loops!

```

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!

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.

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).

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.

Generic Operations on Containers

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

- 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**

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

- reversed(c)** → inverted 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
    
```

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)
d.keys()
d.values()
d.items()
d.pop(key[, default])
d.popitem()
d.get(key[, default])
d.setdefault(key[, default])
    
```

update/add associations
iterable views on keys/values/associations
value
(key, value)
value

Operations on Sets

Operators:

- | → union (vertical bar char)
- & → intersection
- ^ → difference/symmetric diff.
- < <= > >= → inclusion relations

Operators also exist as methods.

```

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

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

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
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)
f.close()
f.flush()
f.tell()
    
```

reading

```

f.read()
f.readlines()
f.readline()
f.truncate([size])
f.seek(position[, origin])
    
```

read empty string if end of file
next chars if *n* not specified, read up to end!
list of next lines
next line
resize
position[, origin]

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

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
    
```

Operations on Strings

```

s.startswith(prefix[, start[, end]])
s.endswith(suffix[, start[, end]])
s.strip([chars])
s.count(sub[, start[, end]])
s.index(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

formatting directives → values to format → **str**

```
"model{ } { }".format(x, y, r)
```

"{selection: formatting!conversion}"

Selection:

```

"{:+.2f}".format(45.72793)
2
nom
0.nom
4[key]
0[2]
    
```

Examples:

```

"'+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)

good habit : don't modify loop variable