

Base Types

integer, float, boolean, string, bytes

```
int 783 0 -192 0b010 0o642 0xFF3
      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
1\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",)

Non modifiable values (immutables) ⚡ expression with only commas → **tuple**
(ordered sequences of chars / bytes)
- key containers, no a priori order, fast key access, each key is unique
 - dict** {"key": "value"} dict(a=3, b=4, k="v")
 - (key/value associations) {1: "one", 3: "three", 2: "two", 3.14: "pi"}
 - 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

- evaluation of right side expression value
- 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
```

Conversions

type (expression)

can specify integer number base in 2nd parameter
truncate decimal part

```
int("15") → 15
int("3f", 16) → 63
int(15.56) → 15
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 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 block 2...
  │ ...
  │
  └ next statement after block 1
```

⚡ 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

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

angles in radians
from math import sin, pi...
sin(pi/4) → 0.707...
cos(2*pi/3) → -0.4999...
sqrt(81) → 9.0 √
log(e2) → 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(...)

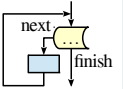
Errors processing:
try:
→ normal processing block
except Exception as e:
→ error processing block

⚡ **finally** block for final processing in all cases.

```

normal processing block
┌ raise X()
└ error processing block
  └ finally block for final processing in all cases.
```

Iterative Loop Statement



<> ^ = + - 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