

Python Cheat Sheet

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" \n: escaped new line
'l\m' l: escaped
Multiline string: """X\tY\tZ
1\t2\t3"""
l: escaped tab
bytes b"toto\xfe\775"
👉 **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
© 8y 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 *a,b=seq
unpacking of sequence in 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

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)
str bytes (ordered sequences of chars / bytes)
• 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}
👉 keys=hashable values (basetypes, immutables...) frozen set immutable set

Conversions

type(expression)
int("15") → 15
int("3f",16) → 63
can specify integer number base in 2nd parameter
int(15.56) → 15 truncate decimal value
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)
lst=[10, 20, 30, 40, 50] positive slice 0 1 2 3 4 5
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

Maths

floating numbers... approximated values
Operators: + - * / // % **
Priority(...)
@ → 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
modules math, statistics, random, decimal, fractions, numpy, etc. (cf. doc)

Statements Blocks

parent statement:
statement block 1...
...
parent statement:
statement block 2...
next statement after block 1
Indentation is mandatory

Conditional Statement

statement block executed only if
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.
👉 with a var x:
if bool(x)==True: ⇔ if x:
if bool(x)==False: ⇔ if not x:

Exceptions on Errors

Signaling an error:
raise ExcClass(...)
Errors processing:
try:
normal processing block
except Exception as e:
error processing block

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 condition with a least one variable value (here i)
while i<=100:
 s=s+i**2
 i = i + 1
print("sum:",s)

Loop Control

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

Display

print("v=",3,"cm :",x,"",y+4)
items to display : literal values, variables, expressions
print options:
□ sep=" " item 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 ci 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)→ 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

Iterative Loop Statement

statements block executed for each item of a container or iterator
for var in sequence:
 statement block
Go over sequence's values
s = "Some text" - initializations before the loop
cnt = 0
loop variable, assignment managed by for statement
for c in s: Algo: count number of e in the string.
 if c == "e":
 cnt = cnt + 1
print("found",cnt,"e")
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)): Algo: limit values greater than 15,
 val = lst[idx] memorizing of lost values.
 if val > 15:
 lost.append(val)
 lst[idx] = 15
print("modif:", lst, "-lost:", lost)
Go simultaneously over sequence's index and values:
for idx,val in enumerate(lst):

Operations on Lists

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
lst.sort() lst.reverse()	sort / reverse liste in place

Operations on Dictionaries

d[key]=value
d[key]→ value
d.clear()
del d[key]
d.update(d2)
d.keys() →iterable views on
d.values()
d.items()
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()

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)

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)

Function Call

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

☞ 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")
opening mode on disk: 'r' read, 'w' write, 'a' append
f.read([n]) → next chars
if n not specified, read up to end !
f.readlines([n]) → list of next lines
f.readline() → next line
f.write("coucou") f.writelines(list of lines)
f.close() ☞ dont forget to close the file after use !
f.flush() write cache f.truncate([size]) resize
f.tell()→position f.seek(position[,origin])