Python Cheat Sheet: Keywords

"A puzzle a day to learn, code, and play" → Visit finxter.com

Keyword	Description	Code example
`False`, `True`	Data values from the data type Boolean	`False == (1 > 2), True == (2 > 1)`

`and`, `or`, `not`	Logical operators:
	`(x and y)` → both x and y must be True
	`(x or y)` → either x or y must be True
	`(not x)` → x must be false
		`x, y = True, False`
		`(x or y) == True       # True`
		`(x and y) == False     # True`
		`(not y) == True        # True`

`break`	Ends loop prematurely	`while(True):`
		`   break # no infinite loop`
		`print("hello world")`

`continue`	Finishes current loop iteration	`while(True):`
		`  continue`
		`  print("43") # dead code`

`class`
 	
`def`
	Defines a new class → a real-world concept (object oriented programming)
	
	Defines a new function or class method. For latter, first parameter ("self") points to the class object. When calling class method, first parameter is implicit.
		`class Beer:`
		`   def __init__(self):`
		`       self.content = 1.0`
		`   def drink(self):`
		`       self.content = 0.0`
		
		`becks = Beer() # constructor - create class`
		`becks.drink() # beer empty: b.content == 0`

`if`, `elif`, `else`	Conditional program execution: program starts with "if" branch, tries the "elif" branches, and finishes with "else" branch (until one branch evaluates to True).	`x = int(input("your value: "))`
		`if x > 3: print("Big")`
		`elif x == 3: print("Medium")`
		`else: print("Small")`

`for`, `while`	`# For loop declaration`
	`for i in [0,1,2]:`
	`   print(i)`
		`# While loop - same semantics`
		`j = 0`
		`while j < 3:`
		`   print(j)`
		`   j = j + 1`

`in`	Checks whether element is in sequence	`42 in [2, 39, 42] # True`

`is`	Checks whether both elements point to the same object	`y = x = 3`
		`x is y # True`
		`[3] is [3] # False`

`None`	Empty value constant	`def f():`
		`   x = 2`
		`f() is None # True`

`lambda`	Function with no name (anonymous function)	`(lambda x: x + 3)(3) # returns 6`

`return`	Terminates execution of the function and passes the flow of execution to the caller. An optional value after the return keyword specifies the function result.	`def incrementor(x):`
		`   return x + 1`
		`incrementor(4) # returns 5`

finxter

Python Cheat Sheet: Basic Data Types

“A puzzle a day to learn, code, and play” → Visit finxter.com

	Description	Example
Boolean	The Boolean data type is a truth value, either True or False. The Boolean operators ordered by priority: not x → “if x is False, then x, else y” x and y → “if x is False, then x, else y” x or y → “if x is False, then y, else x” These comparison operators evaluate to True: 1 < 2 and 0 <= 1 and 3 > 2 and 2 >=2 and 1 == 1 and 1 != 0 # True	## 1. Boolean Operations x, y = True, False print(x and not y) # True print(not x and y or x) # True ## 2. If condition evaluates to False if None or 0 or 0.0 or '' or [] or {} or set(): # None, 0, 0.0, empty strings, or empty # container types are evaluated to False print("Dead code") # Not reached

Integer, Float	An integer is a positive or negative number without floating point (e.g. 3). A float is a positive or negative number with floating point precision (e.g. 3.14159265359).
	
	The ‘//’ operator performs integer division. The result is an integer value that is rounded toward the smaller integer number (e.g. 3 // 2 == 1).
		## 3. Arithmetic Operations
		x, y = 3, 2
		print(x + y) # = 5
		print(x - y) # = 1
		print(x * y) # = 6
		print(x / y) # = 1.5
		print(x // y) # = 1
		print(x % y) # = 1s
		print(-x) # = -3
		print(abs(-x)) # = 3
		print(int(3.9)) # = 3
		print(float(3)) # = 3.0
		print(x ** y) # = 9

String	Python Strings are sequences of characters.
	
	The four main ways to create strings are the following.
	
	1. Single quotes
	'Yes'
	2. Double quotes
	"Yes"
	3. Triple quotes (multi-line)
	"""Yes
	We Can"""
	4. String method
	str(5) == '5' # True
	5. Concatenation
	"Ma" + "hatma" # 'Mahatma'
	
	These are whitespace characters in strings.
	● Newline \n
	● Space \s
	● Tab \t
		## 4. Indexing and Slicing
		s = "The youngest pope was 11 years old"
		print(s[0]) # 'T'
		print(s[1:3]) # 'he'
		print(s[-3:-1]) # 'ol'
		print(s[-3:]) # 'old'
		x = s.split() # creates string array of words
		print(x[-3] + " " + x[-1] + " " + x[2] + "s")
		                  # '11 old popes'
		
		## 5. Most Important String Methods
		y = " This is lazy\t\n "
		print(y.strip()) # Remove Whitespace: 'This is lazy'
		print("DrDre".lower()) # Lowercase: 'drdre'
		print("attention".upper()) # Uppercase: 'ATTENTION'
		print("smartphone".startswith("smart")) # True
		print("smartphone".endswith("phone")) # True
		print("another".find("other")) # Match index: 2
		print("cheat".replace("ch", "m")) # 'meat'
		print(','.join(["F", "B", "I"])) # 'F,B,I'
		print(len("Rumpelstiltskin")) # String length: 15
		print("ear" in "earth") # Contains: True

finxter

Python Cheat Sheet: Complex Data Types

"A puzzle a day to learn, code, and play" → Visit finxter.com

	Description	Example

List	A container data type that stores a sequence of elements. Unlike strings, lists are mutable: modification possible.	`l = [1, 2, 2]`
		`print(len(l)) # 3`

Adding elements	Add elements to a list with (i) append, (ii) insert, or (iii) list concatenation.
	The append operation is very fast.
		`[1, 2, 2].append(4) # [1, 2, 2, 4]`
		`[1, 2, 4].insert(2,2) # [1, 2, 2, 4]`
		`[1, 2, 2] + [4] # [1, 2, 2, 4]`

Removal	Removing an element can be slower.	`[1, 2, 2, 4].remove(1) # [2, 2, 4]`
Reversing	This reverses the order of list elements.	`[1, 2, 3].reverse() # [3, 2, 1]`
Sorting	Sorts a list. The computational complexity of sorting is linear in the no. list elements.	`[2, 4, 2].sort() # [2, 2, 4]`

Indexing	Finds the first occurence of an element in the list & returns its index. Can be slow as the whole list is traversed.	`[2, 2, 4].index(2) # index of element 4 is "0"`
		`[2, 2, 4].index(2,1) # index of element 2 after pos 1 is "1"`

Stack	Python lists can be used intuitively as stacks via the two list operations append() and pop().	`stack = [3]`
		`stack.append(42) # [3, 42]`
		`stack.pop() # 42 (stack: [3])`
		`stack.pop() # 3 (stack: [])`

Set	A set is an unordered collection of unique elements ("at-most-once").	`basket = {'apple', 'eggs', 'banana', 'orange'}`
		`same = set(['apple', 'eggs', 'banana', 'orange'])`

Dictionary	The dictionary is a useful data structure for storing (key, value) pairs.	`calories = {'apple' : 52, 'banana' : 89, 'choco' : 546}`

Reading and writing elements	Read and write elements by specifying the key within the brackets. Use the keys() and values() functions to access all keys and values of the dictionary.	`print(calories['apple'] < calories['choco']) # True`
		`calories['cappu'] = 74`
		`print(calories['banana'] < calories['cappu']) # False`
		`print('apple' in calories.keys()) # True`
		`print(52 in calories.values()) # True`

Dictionary Looping	You can access the (key, value) pairs of a dictionary with the `items()` method.	`for k, v in calories.items():`
		`       print(k) if v > 500 else None # 'chocolate'`

Membership operator	Check with the 'in' keyword whether the set, list, or dictionary contains an element. Set containment is faster than list containment.	`basket = {'apple', 'eggs', 'banana', 'orange'}`
		`print('eggs' in basket) # True`
		`print('mushroom' in basket) # False`

List and Set Comprehension	List comprehension is the concise Python way to create lists. Use brackets plus an expression, followed by a for clause. Close with zero or more for or if clauses.
	
	Set comprehension is similar to list comprehension.
		`# List comprehension`
		`l = [('Hi ' + x) for x in ['Alice', 'Bob', 'Pete']]`
		`print(l) # ['Hi Alice', 'Hi Bob', 'Hi Pete']`
		`l2 = [x * y for x in range(3) for y in range(3) if x>y]`
		`print(l2) # [0, 0, 2]`
		`# Set comprehension`
		`squares = { x**2 for x in [0,2,4] if x < 4 } # {0, 4}`

finxter

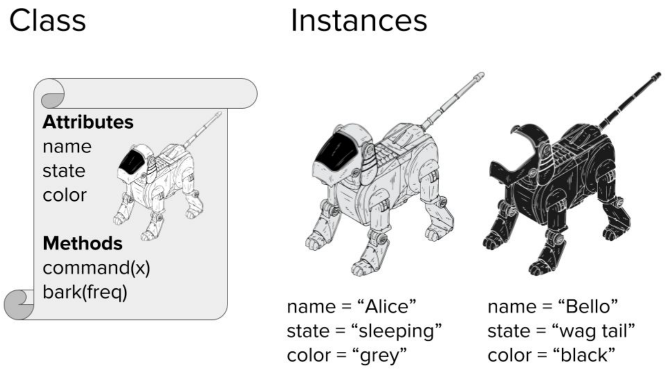
Python Cheat Sheet: Classes

"A puzzle a day to learn, code, and play" → Visit finxter.com

Description

Classes

A class encapsulates data and functionality: data as attributes, and functionality as methods. It is a blueprint for creating concrete instances in memory.



Instance

You are an instance of the class human. An instance is a concrete implementation of a class: all attributes of an instance have a fixed value. Your hair is blond, brown, or black--but never unspecified.

Each instance has its own attributes independent of other instances. Yet, class variables are different. These are data values associated with the class, not the instances. Hence, all instance share the same class variable `species` in the example.

Self

The first argument when defining any method is always the `self` argument. This argument specifies the instance on which you call the method.

`self` gives the Python interpreter the information about the concrete instance. To *define* a method, you use `self` to modify the instance attributes. But to *call* an instance method, you do not need to specify `self`.

Creation

You can create classes "on the fly" and use them as logical units to store complex data types.

```python
class Employee():
        pass
employee = Employee()
employee.salary = 122000
employee.firstname = "alice"
employee.lastname = "wonderland"

print(employee.firstname + " "
        + employee.lastname + " "
        + str(employee.salary) + "$")
# alice wonderland 122000$
```

Example

```python
class Dog:
        """ Blueprint of a dog """

        # class variable shared by all instances
        species = ["canis lupus"]

        def __init__(self, name, color):
                self.name = name
                self.state = "sleeping"
                self.color = color

        def command(self, x):
                if x == self.name:
                        self.bark(2)
                elif x == "sit":
                        self.state = "sit"
                else:
                        self.state = "wag tail"

        def bark(self, freq):
                for i in range(freq):
                        print("[" + self.name
                                + "]: Woof!")


bello = Dog("bello", "black")
alice = Dog("alice", "white")

print(bello.color) # black
print(alice.color) # white

bello.bark(1) # [bello]: Woof!

alice.command("sit")
print("[alice]: " + alice.state)
# [alice]: sit

bello.command("no")
print("[bello]: " + bello.state)
# [bello]: wag tail

alice.command("alice")
# [alice]: Woof!
# [alice]: Woof!

bello.species += ["wulf"]
print(len(bello.species)
        == len(alice.species)) # True (!)
```

finxter

Python Cheat Sheet: Functions and Tricks

"A puzzle a day to learn, code, and play" ➔ Visit finxter.com

		Description	Example	Result
ADVANCED FUNCTIONS	map(func, iter)	Executes the function on all elements of the iterable	list(map(lambda x: x[0], ['red', 'green', 'blue']))	['r', 'g', 'b']
	map(func, i1, ..., ik)	Executes the function on all k elements of the k iterables	list(map(lambda x, y: str(x) + ' ' + y + 's' , [0, 2, 2], ['apple', 'orange', 'banana']))	['0 apples', '2 oranges', '2 bananas']
	string.join(iter)	Concatenates iterable elements separated by string	' marries '.join(list(['Alice', 'Bob']))	'Alice marries Bob'
	filter(func, iterable)	Filters out elements in iterable for which function returns False (or 0)	list(filter(lambda x: True if x>17 else False, [1, 15, 17, 18]))	[18]
	string.strip()	Removes leading and trailing whitespaces of string	print(" \n \t 42 \t ".strip())	42
	sorted(iter)	Sorts iterable in ascending order	sorted([8, 3, 2, 42, 5])	[2, 3, 5, 8, 42]
	sorted(iter, key=key)	Sorts according to the key function in ascending order	sorted([8, 3, 2, 42, 5], key=lambda x: 0 if x==42 else x)	[42, 2, 3, 5, 8]
	help(func)	Returns documentation of func	help(str.upper())	'... to uppercase.'
	zip(i1, i2, ...)	Groups the i-th elements of iterators i1, i2, ... together	list(zip(['Alice', 'Anna'], ['Bob', 'Jon', 'Frank']))	[('Alice', 'Bob'), ('Anna', 'Jon')]
	Unzip	Equal to: 1) unpack the zipped list, 2) zip the result	list(zip(*[('Alice', 'Bob'), ('Anna', 'Jon')]))	[('Alice', 'Anna'), ('Bob', 'Jon')]
	enumerate(iter)	Assigns a counter value to each element of the iterable	list(enumerate(['Alice', 'Bob', 'Jon']))	[(0, 'Alice'), (1, 'Bob'), (2, 'Jon')]
TRICKS	python -m http.server <P>	Want to share files between PC and phone? Run this command in PC's shell. <P> is any port number 0–65535. Type < IP address of PC>:<P> in the phone's browser. You can now browse the files in the PC directory.		
	Read comic	import antigravity	Open the comic series xkcd in your web browser	
	Zen of Python	import this	'...Beautiful is better than ugly. Explicit is ...'	

	Swapping numbers	Swapping variables is a breeze in Python. No offense, Java!	a, b = 'Jane', 'Alice'
			a, b = b, a
				a = 'Alice'
				b = 'Jane'

	Unpacking arguments	Use a sequence as function arguments via asterisk operator *. Use a dictionary (key, value) via double asterisk operator **	def f(x, y, z): return x + y * z
			f(*[1, 3, 4])
			f(**{'z' : 4, 'x' : 1, 'y' : 3})
				
				13
				13

	Extended Unpacking	Use unpacking for multiple assignment feature in Python	a, *b = [1, 2, 3, 4, 5]	a = 1
				b = [2, 3, 4, 5]

	Merge two dictionaries	Use unpacking to merge two dictionaries into a single one	x={'Alice' : 18}
			y={'Bob' : 27, 'Ann' : 22}
			z = {**x,**y}
				z = {'Alice': 18, 'Bob': 27, 'Ann': 22}

finxter

Python Cheat Sheet: 14 Interview Questions

"A puzzle a day to learn, code, and play" ➜ Visit finxter.com

Question	Code	Question	Code

Check if list contains integer x	`l = [3, 3, 4, 5, 2, 111, 5]`
	`print(111 in l) # True`
		Get missing number in [1...100]	`def get_missing_number(lst):`
			`   return set(range(lst[len(lst)-1])[1:]) - set(l)`
			`l = list(range(1,100))`
			`l.remove(50)`
			`print(get_missing_number(l)) # 50`

Find duplicate number in integer list	`def find_duplicates(elements):`
	`   duplicates, seen = set(), set()`
	`   for element in elements:`
	`      if element in seen:`
	`          duplicates.add(element)`
	`      seen.add(element)`
	`   return list(duplicates)`
		Compute the intersection of two lists	`def intersect(lst1, lst2):`
			`   res, lst2_copy = [], lst2[:]`
			`   for el in lst1:`
			`       if el in lst2_copy:`
			`           res.append(el)`
			`           lst2_copy.remove(el)`
			`   return res`

Check if two strings are anagrams	`def is_anagram(s1, s2):`
	`   return set(s1) == set(s2)`
	`print(is_anagram("elvis", "lives")) # True`
		Find max and min in unsorted list	`l = [4, 3, 6, 3, 4, 888, 1, -11, 22, 3]`
			`print(max(l)) # 888`
			`print(min(l)) # -11`

Remove all duplicates from list	`lst = list(range(10)) + list(range(10))`
	`lst = list(set(lst))`
	`print(lst)`
	`# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
		Reverse string using recursion	`def reverse(string):`
			`    if len(string)<=1: return string`
			`    return reverse(string[1:])+string[0]`
			`print(reverse("hello")) # olleh`

Find pairs of integers in list so that their sum is equal to integer x	`def find_pairs(l, x):`
	`   pairs = []`
	`   for (i, el_1) in enumerate(l):`
	`      for (j, el_2) in enumerate(l[i+1:]):`
	`         if el_1 + el_2 == x:`
	`            pairs.append((el_1, el_2))`
	`   return pairs`
		Compute the first n Fibonacci numbers	`a, b = 0, 1`
			`n = 10`
			`for i in range(n):`
			`    print(b)`
			`    a, b = b, a+b`
			`# 1, 1, 2, 3, 5, 8, ...`

Check if a string is a palindrome	`def is_palindrome(phrase):`
	`    return phrase == phrase[::-1]`
	`print(is_palindrome("anna")) # True`
		Sort list with Quicksort algorithm	`def qsort(L):`
			`    if L == []: return []`
			`    return qsort([x for x in L[1:] if x< L[0]]) + L[0:1] +`
			`qsort([x for x in L[1:] if x>=L[0]])`
			`lst = [44, 33, 22, 5, 77, 55, 999]`
			`print(qsort(lst))`
			`# [5, 22, 33, 44, 55, 77, 999]`

Use list as stack, array, and queue	`# as a list ...`
	`l = [3, 4]`
	`l +=  [5, 6] # l = [3, 4, 5, 6]`
	
	`# ... as a stack ...`
	`l.append(10) # l = [4, 5, 6, 10]`
	`l.pop() # l = [4, 5, 6]`
	
	`# ... and as a queue`
	`l.insert(0, 5) # l = [5, 4, 5, 6]`
	`l.pop() # l = [5, 4, 5]`
		Find all permutations of string	`def get_permutations(w):`
			`    if len(w)<=1:`
			`        return set(w)`
			`    smaller = get_permutations(w[1:])`
			`    perms = set()`
			`    for x in smaller:`
			`        for pos in range(0,len(x)+1):`
			`            perm = x[:pos] + w[0] + x[pos:]`
			`            perms.add(perm)`
			`    return perms`
			`print(get_permutations("nan"))`
			`# {'nna', 'ann', 'nan'}`

Python Cheat Sheet: List Methods

"*A puzzle a day to learn, code, and play*" → Visit finxter.com

Method	Description	Example

`lst.append(x)`	Appends element `x` to the list `lst`.	`>>> l = []`
		`>>> l.append(42)`
		`>>> l.append(21)`
		`[42, 21]`

`lst.clear()`	Removes all elements from the list lst–which becomes empty.	`>>> lst = [1, 2, 3, 4, 5]`
		`>>> lst.clear()`
		`[]`

`lst.copy()`	Returns a copy of the list `lst`. Copies only the list, not the elements in the list (shallow copy).	`>>> lst = [1, 2, 3]`
		`>>> lst.copy()`
		`[1, 2, 3]`

`lst.count(x)`	Counts the number of occurrences of element `x` in the list `lst`.	`>>> lst = [1, 2, 42, 2, 1, 42, 42]`
		`>>> lst.count(42)`
		`3`
		`>>> lst.count(2)`
		`2`

`lst.extend(iter)`	Adds all elements of an iterable `iter` (e.g. another list) to the list `lst`.	`>>> lst = [1, 2, 3]`
		`>>> lst.extend([4, 5, 6])`
		`[1, 2, 3, 4, 5, 6]`

`lst.index(x)`	Returns the position (index) of the first occurrence of value `x` in the list `lst`.	`>>> lst = ["Alice", 42, "Bob", 99]`
		`>>> lst.index("Alice")`
		`0`
		`>>> lst.index(99, 1, 3)`
		`ValueError: 99 is not in list`

`lst.insert(i, x)`	Inserts element `x` at position (index) `i` in the list `lst`.	`>>> lst = [1, 2, 3, 4]`
		`>>> lst.insert(3, 99)`
		`[1, 2, 3, 99, 4]`

`lst.pop()`	Removes and returns the final element of the list `lst`.	`>>> lst = [1, 2, 3]`
		`>>> lst.pop()`
		`3`
		`>>> lst`
		`[1, 2]`

`lst.remove(x)`	Removes and returns the first occurrence of element `x` in the list `lst`.	`>>> lst = [1, 2, 99, 4, 99]`
		`>>> lst.remove(99)`
		`>>> lst`
		`[1, 2, 4, 99]`

`lst.reverse()`	Reverses the order of elements in the list `lst`.	`>>> lst = [1, 2, 3, 4]`
		`>>> lst.reverse()`
		`>>> lst`
		`[4, 3, 2, 1]`

`lst.sort()`	Sorts the elements in the list `lst` in ascending order.	`>>> lst = [88, 12, 42, 11, 2]`
		`>>> lst.sort()`
		`# [2, 11, 12, 42, 88]`
		`>>> lst.sort(key=lambda x: str(x)[0])`
		`# [11, 12, 2, 42, 88]`

finxter

Python 3 Cheat Sheet

Latest version on :
https://perso.limsi.fr/pointal/python:memento

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
                        ×10⁻⁶
  bool True  False
   str "One\nTwo"          Multiline string:
       escaped new line      """X\tY\tZ
        'I\'m'               1\t2\t3"""
        escaped '              escaped tab
bytes b"toto\xfe\775"
        hexadecimal  octal                ☝ 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 comas → tuple
        str bytes  (ordered sequences of chars / bytes)           ""
                                                                  b""
```

- **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}             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*
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  } unpacking of sequence in
*a,b=seq  } item and list
x+=3        increment ⇔ x=x+3          and
x-=2        decrement ⇔ x=x-2          *=
x=None      « undefined » constant value  /=
del x       remove name x               %=
                                        …
```

Conversions

```
int("15") → 15                 type(expression)
int("3f",16) → 63              can specify integer number base in 2nd parameter
int(15.56) → 15                truncate decimal part
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

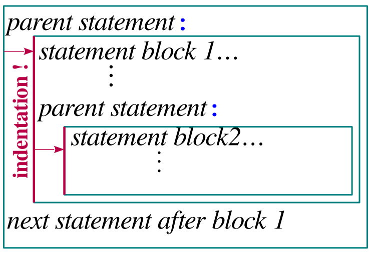
Comparisons : `< > <= >= == !=`
(boolean results)   ≤ ≥ = ≠

```
a and b   logical and   both simulta-
                        -neously
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 block2…
            ⁝
next statement after block 1
```

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

Modules/Names Imports

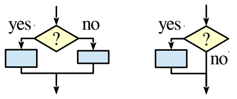
module `truc`⇔*file* `truc.py`

```
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*   *angles in radians*

```
Operators: + - * / // % **
Priority (…)   × ÷  ↑    ↑    aᵇ
              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*

```
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)*

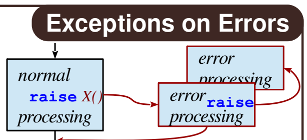
Exceptions on Errors

Signaling an error:

```
    raise ExcClass(…)
```

Errors processing:

```
try:
    normal processing block
except Exception as e:
    error processing block
```

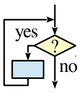


☝ `finally` *block for final processing in all cases.*

Conditional Loop Statement

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

```
while logical condition:
    statements block
```



```
s = 0
i = 1
```
*initializations **before** the loop*

condition with a least one variable value (here `i`)

```
while i <= 100:
    s = s + i**2
    i = i + 1
print("sum:",s)
```
☝ *make condition variable change !*

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

☝ *beware of infinite loops!*

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=" "` 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)`

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

`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)`→ *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

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*

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()`

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…)

cf. modules `os`, `os.path` and `pathlib`

opening **mode**

- □ `'r'` read
- □ `'w'` write
- □ `'a'` append
- □ …`'+'` `'x'` `'b'` `'t'`

encoding of chars for *text files*: `utf8` `ascii` `latin1` …

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])`

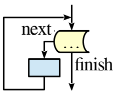
Very common: opening with a guarded block (automatic closing) and reading loop on lines of a text file:

```
with open(…) as f:
    for line in f :
        # processing of line
```

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):`

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*

Function Definition

function name (identifier)

named parameters

```
def fct(x,y,z):
    """documentation"""
    # statements block, res computation, etc.
    return res
```

`fct`

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

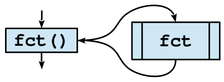
`r = fct(3,i+2,2*i)`

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)`

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)