Basics of Programming in Python

Programmazione Avanzata 2025-26

Topics

- Comments
- Variables
- Functions (basics): using functions, modules and libraries
- Strings
- Decisions
- Iterations
- Lists (basics): creating lists, adding, and iterating on list elements
- Functions (advanced): defining functions, scope, visibility
- Files
- Exceptions
- Lists (advanced) and other data structures, sorting

Comments

- It is a good practice to add comments at the beginning of each program file to clarify contents and goals
- Comments will be ignored by the Python interpreter, they are meant for humans (programmers)
- Single line comments
  - Using character #
- Multi-line comments
  - Using characters `"""`, as multiline strings

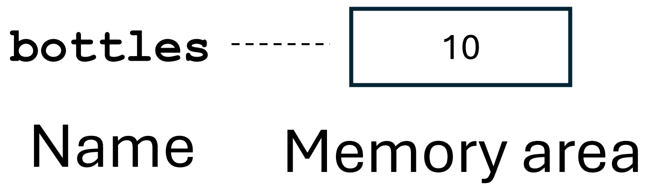
Variables

- Definition and assignment
- Name and memory
- Data types
- Naming conventions
- Constants
- Arithmetic operators

Defining variables and assigning values

- A variable is a memory area identified by (reachable via) a name, which makes reference to a specific value
- A variable is defined by telling to the Python interpreter
  - Its name (later used for referring to it)
  - Its initial value
- To set the initial value (or change it afterward), an assignment instruction is used, leveraging the = operator

```
bottles = 10 # defines and initial. the variable
```

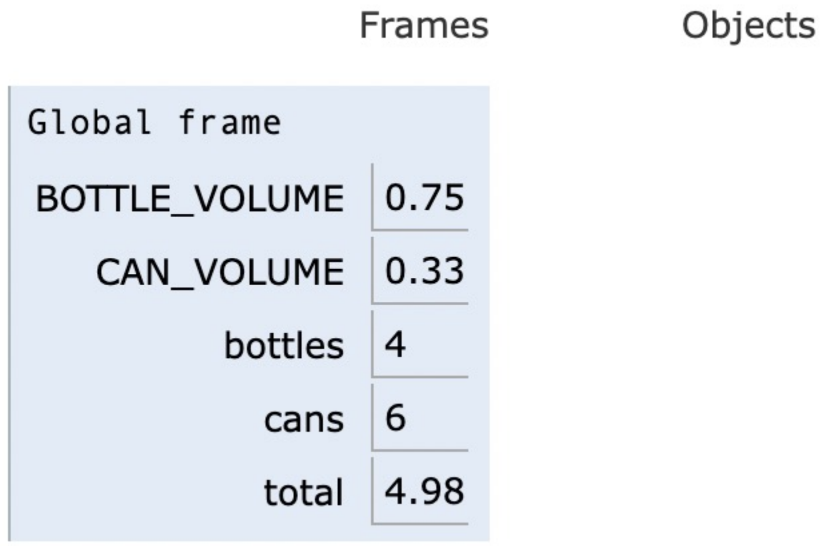


Visualizing variables

```
# Constants
BOTTLE_VOLUME = 0.75
CAN_VOLUME = 0.33

# Data
bottles = 4
cans = 6

# Computation
total = 0
total = bottles * BOTTLE_VOLUME
total = total + cans * CAN_VOLUME
```



Global frame

BOTTLE_VOLUME	0.75
CAN_VOLUME	0.33
bottles	4
cans	6
total	4.98

http://pythontutor.com/

Types

- In many programming languages, each variable is assigned a data type (so that proper memory is allocated)
  - There are different primitive data types (also known as basic or built-in types) in Python: integer (**`int`**), floating point (**`float`**), boolean (**`bool`**), string (**`str`**), none (**`None`**)
  - Python is a dynamically typed language, meaning that variable type does not need to be defined explicitly (at compile time)
  - The type is determined dynamically at runtime based on the assigned value (i.e., it is linked to value, not to variable)

```
cansPerPack = 6 # int
canVolume = 12.0 # float
```

Undefined variables

- Variables need to be defined before using them the first time (in a previous line of code)

```
canVolume = 12 * literPerOunce
literPerOunce = 0.0296
>>> NameError: name 'literPerOunce' is not defined
```

Changing data type

- A variable can be assigned different types at different times during program lifetime
  - `taxRate = 5 # holding an integer value`
  - `taxRate = 5.5 # now a float`
  - `taxRate = "Non-taxable" # now a string`
- Once a variable has been assigned a given value, Python checks that operations performed on it are valid based on actual value
- Programmers need to recall the variable type

Naming conventions

- Variables names shall begin with a letter or underscore character _
  - Next characters can be letters, numbers, or _
  - Other symbols (like `?`, `%`, ...) or spaces are not allowed
  - Words in names are typically separated following conventions
    - Snake_case: using _ to separate words (preferred)
    - camelCase: using capitalized initials to mark word boundaries (typical of OO programming languages)
  - “Reserved” words of Python language must be avoided
- Programmers should choose meaningful names

Constants

- In Python, a constant is a variable whose value should not be modified once the initial value has been assigned
- Conventionally, all capital letters are used (easier for the programmers to distinguish them, e.g., in expressions)
- Python do not block changes to constants, though it does not mean it is wise doing it

```
BOTTLE_VOLUME = 2
MAX_SIZE = 100
taxRate = 5
```

Arithmetic operators

- Basic arithmetic operators are:
  - Addition: `+`
  - Substraction: `-`
  - Multiplication: `*`
  - Division: `/`
  - Power: `**`
  - Integer division: `//` (remainder is obtained using `%`)
- Although precedence rules are defined (following algebra), the suggestion is to use parentheses when writing expressions

```
c = b * ((1 + r / 100) ** n)
```

Combining types in expressions

- When combining integer and floating point numbers in a mathematical expression, the result will be a number in floating point
- Combining numbers and strings in a mathematical expression will lead to an error

Functions (basics)

- Goal of functions: code factorization
- Parameters
- Return value
- Libraries and modules
- Built-in functions and standard library
- Conversion functions

Functions

- A function is used to "factorize" a group of instructions that address a particular need / solve a given problem
- When needed, it is possible to invoke (call) that function, rather than writing all the single instructions (and repeating them all the times)
  - When invoking a function, it is necessary to pass as arguments the correct number of parameters
- Most of the functions return a value: when work is over, release the control (and the value) where they were invoked
  - For instance, the invocation of function **`abs(-173)`**, which receives as argument number **`-173`**, returns the value **`173`** that, in turn, may be printed using function **`print(abs(-173))`**

Libraries

- A library in Python is a collection of code (functions, constants, data types, ...) written and compiled by third parties, which is ready to use in a program
  - It is always recommended to check documentation accompanying the library
- A standard library is a library that is considered as part of the language and is included in any Python development environment
  - https://docs.python.org/3/library/index.html

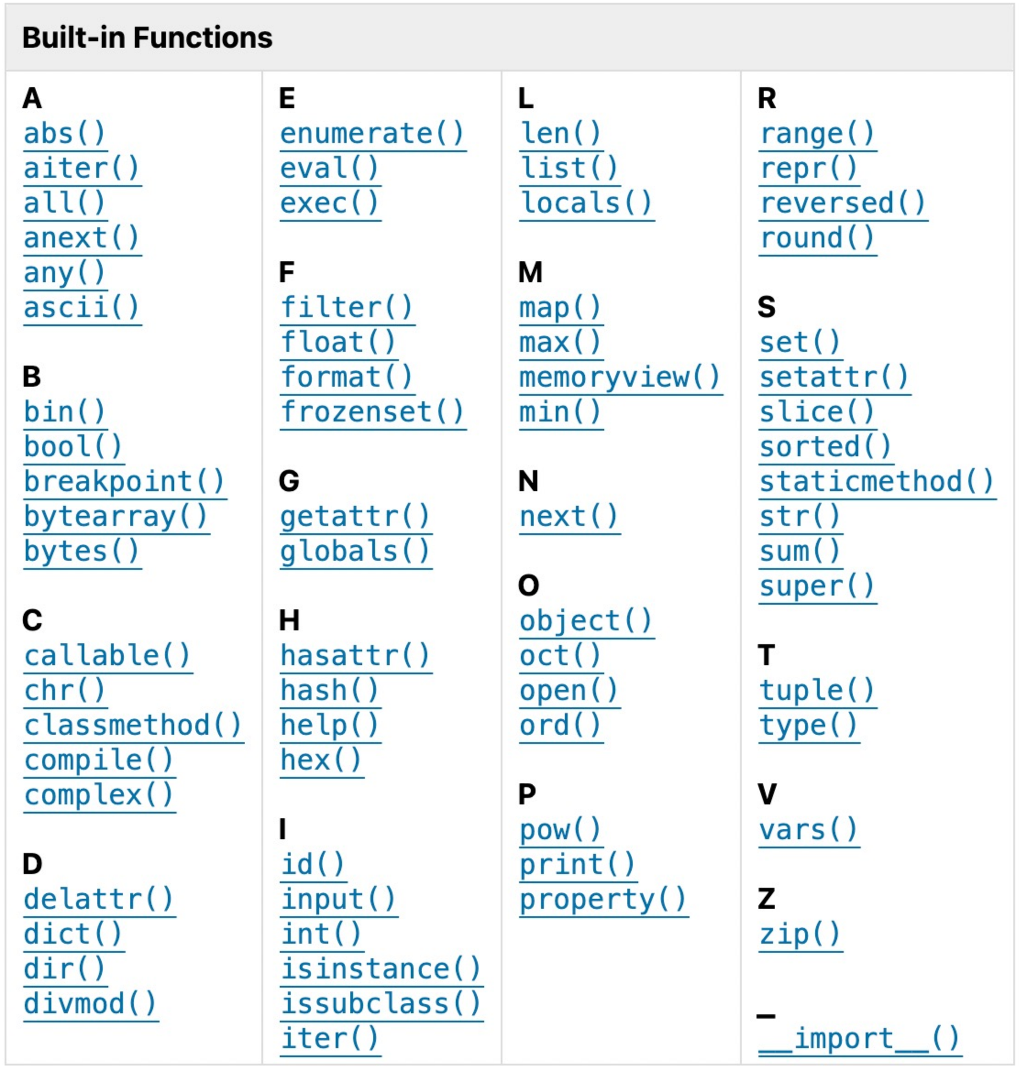
Modules

- Libraries, including the standard library, are organized in modules
- Related functions and data types are typically grouped in the same modules
- Functions defined in a module need to be explicitly loaded (imported) in a program for being used in it though the keyword **`import`**

Libraries, modules, and functions

- Predefined (built-in) functions
  - Always available, like **`print()`**, **`len()`**, **`range()`**
  - https://docs.python.org/3/library/functions.html
- Standard library
  - Part of each Python installation
  - Organized in modules, each containing multiple functions
  - Before using a function, the module defining it must be imported
- Additional libraries
  - Not part of Python installation
  - Need to be downloaded and installed in the project (using IDE or command line), e.g., from https://pypi.org/ (over 200k modules)
  - Once installed, they can be imported and their functions can be used

Predefined (built-in) functions



Built-in Functions

A	**E**	**L**	**R**
abs()	enumerate()	len()	range()
aiter()	eval()	list()	repr()
all()	exec()	locals()	reversed()
anext()			round()
any()	**F**	**M**	
ascii()	filter()	map()	**S**
	float()	max()	set()
B	format()	memoryview()	setattr()
bin()	frozenset()	min()	slice()
bool()			sorted()
breakpoint()	**G**	**N**	staticmethod()
bytearray()	getattr()	next()	str()
bytes()	globals()		sum()
		O	super()
C	**H**	object()	
callable()	hasattr()	oct()	**T**
chr()	hash()	open()	tuple()
classmethod()	help()	ord()	type()
compile()	hex()		
complex()		**P**	**V**
	I	pow()	vars()
D	id()	print()	
delattr()	input()	property()	**Z**
dict()	int()		zip()
dir()	isinstance()		
divmod()	issubclass()		**_**
	iter()		__import__()

Functions in the standard library

- To use, e.g., the function **`sqrt()`**, a module of the standard library named **`math`**, which defines it, must be imported beforehand
  - https://docs.python.org/3/library/math.html

```
# First include this statement at the top
from math import sqrt
# Then function can be called as
y = sqrt(x)
```

Importing modules and functions

- There are three modes for performing the import

```
from math import sqrt, sin, cos
# imports listed functions

from math import *
# imports all functions from the module

import math
# imports module and gives access to all
functions: in this case, the name of the
module and . needs to be used when invoking
functions, e.g., math.sqrt(x)
```

Importing modules and functions

- For instance, Python offers a (pseudo) random number generator, that can be helpful to introduce variability in a program, simulation, …

```
from random import random, randint
r = random() # returns a value >= 0 and <1
randint(1, 6) # return an int >=1 and <=6
```

Conversion functions

- Among predefined functions there are mechanisms for performing data type conversions, e.g., between floating point and integer numbers: **`int()`**, **`float()`**
  - **`balance = total + tax`** `# balance: float`
  - **`dollars = int(balance)`** `# dollars: integer`
- Functions above also work on strings
  - Care needed, operation may lead to errors with improper data
- Function **`round(x,d)`**, still among built-in functions, performs a rounding of **`x`** with a precision of **`d`** digits
- Other functios in math: **`floor()`**, **`ceil()`**, **`trunc()`**

Motivations for using modules

- When writing small programs, all the code can be included in a single source file
- When program size starts to grow, though, or when working in a team (or with multiple teams), situation changes and it is advisable to organize the code in separate source files (modules)
- With modularization it is possible to:
  - Cluster related pieces of code, making localization of code and debugging easier
  - Distribute responsibilities and foster collaboration, having programmers or teams working on their own modules

Organizing programs in modules

- Large Python programs typically consist of a main module (the so called “driver module”) and one or more supplementary modules
- The main module contains function **`main()`**
- Supplementary modules contain support functions, constants, variables, etc.

Strings

- Sequences of characters
- Concatenation
- Length
- Accessing characters
- Converting from numbers to strings and vice versa
- String and objects (with methods)
- Input from the console
- Formatted output

Strings

- Strings are sequences of characters (letters, numbers, punctuation marks, spaces, ...)
- In Python, explicit strings (literals) are specified including the sequence of characters into single or double quotation marks

```
print("This is a string.", 'So is this.')
```

- The use of both " and ' makes it easier, e.g., to include these characters as part of a string

```
message = 'He said "Hello"'
```

- Escape sequences (e.g., `\"`, `\\`, `\n`) are ways to include special characters in strings

String concatenation

- A string can be appended at the end of another string with **+**

```
firstName = "Harry"
lastName = "Morgan"
name = firstName + lastName # HarryMorgan
print(”My name is:", name)
```

- The use of symbol **+** for concatenating strings is an example of operator overloading, i.e., operators that have different meanings (behaviors) depending on the actual use (data type)
  - In this case, not an addition

String length

- The number of characters in a string, i.e., the string length, can be obtained using function **`len()`**

```
length = len("World!") # length is 6
```

- A string of length `0` is also called “empty string”, and it is generally written as `""` or `''`

Converting numbers in strings

- The **+** operator works on strings
  - Concatenating a string with a number leads to an error
- Numbers can be converted (first) into strings using function **`str()`**

```
balance = 888.88
balanceAsString = str(balance)
print(balanceAsString)
```

Converting strings in numbers

- When a string contains the representation of a number (integer or floating point), it can be converted into a numerical value using functions **`int()`** and **`float()`**
- Helpful, e.g., to handle user input

```
val = int("1729")
price = float("17.29")
print(val)
print(price)
```

- Functions above may raise a **`ValueError`** if the string does not contain a correct number representation

Repeating strings

- It is possible to generate a string that is the repetition of a string

```
Dashes = "-" * 20
# --------------------
```

- Symbol `*` is another example of overloaded operator

Accessing string characters

- Each character in a string is assigned an integer index
- First character has index `0`, last character index `len(s)-1`
- Operator `[]` returns the character given the index

```
name = "Harry"
first = name[0]
last = name[4]
```

- The use of a wrong (out of string boundaries) index raises an `IndexError`
- In Python, strings are immutable, i.e., cannot be modified after creation (otherwise, a `TypeError` is raised)
  - Though a new string can be assigned to the same variable, when needed

Portions of strings

- Extracting a portion (or slice, substring) of a string **`s`**

```
portion = s[start : stop]
portion = s[4 : 10]
portion = s[ : 6]
portion = s[6 : ]
```

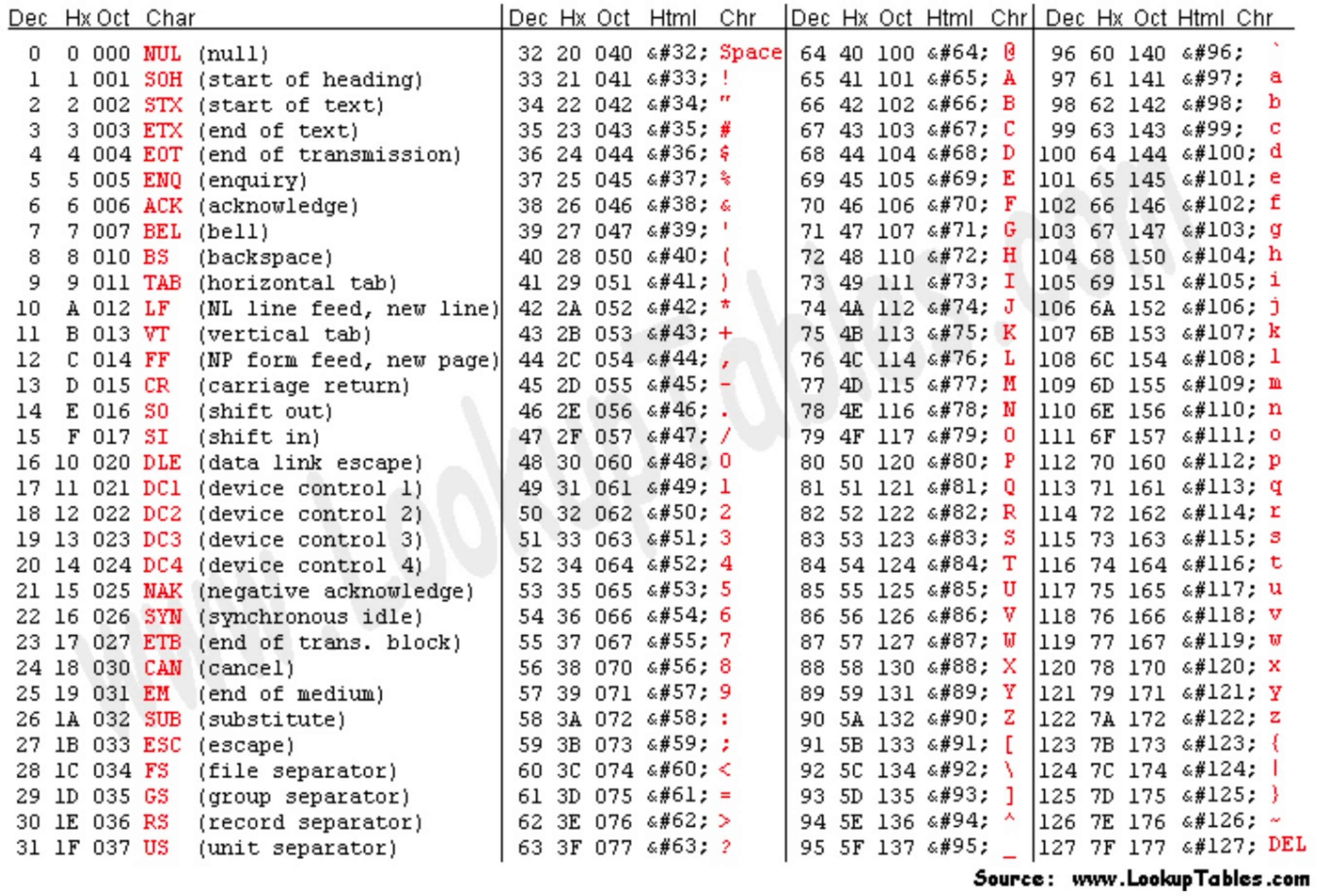
- Extracting a slice with a step

```
portion = s[start : stop : step]
```

Characters

- Characters in Python are stored as integer values, according to ASCII chart in Unicode standard
  - https://home.unicode.org/
  - http://www.unicode.org/charts/
- For instance, letter **`'H'`** has ASCII code **`72`**
- Python offers functions for converting characters
  - **`ord()`**: receives a string representing a Unicode character (e.g., **`'x'`**) and returns an integer representing the unicode code of that character (**`120`**)
  - **`chr()`**: receives the Unicode code of a character (e.g., **`97`**) and returns a string representing the character **`'a'`**

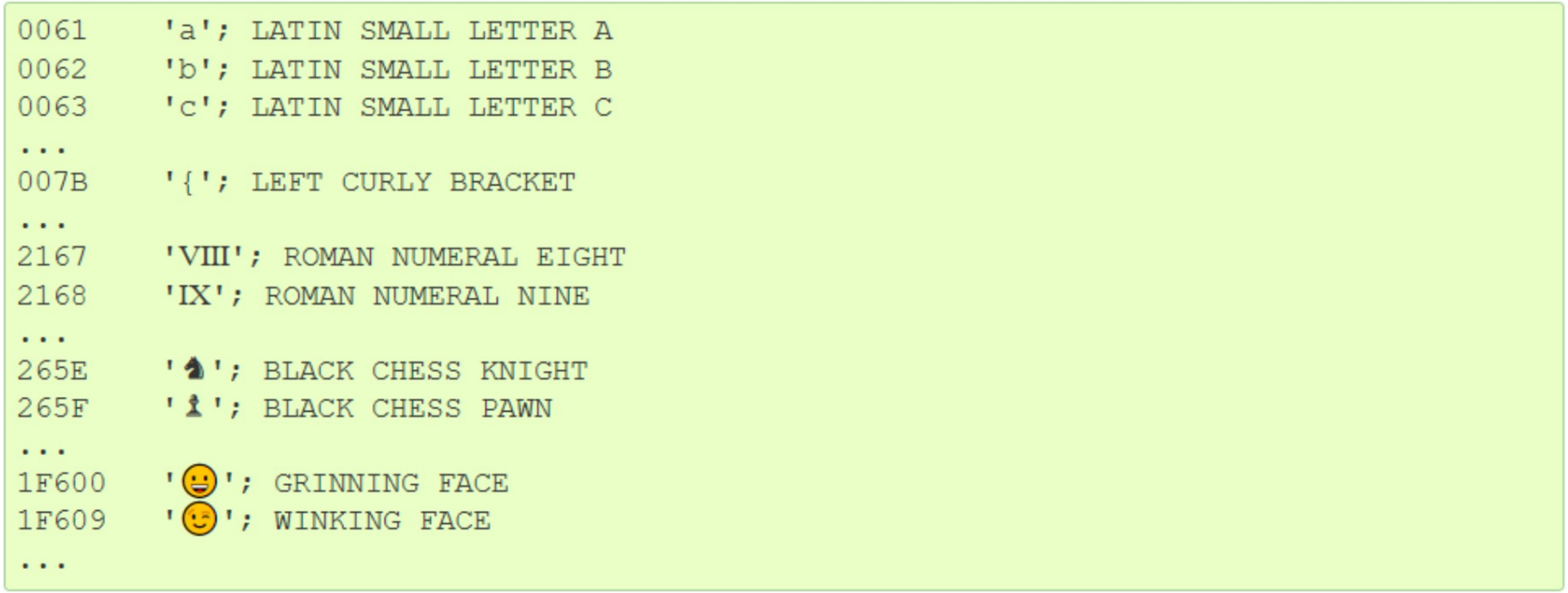
ASCII codes



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Unicode codes



```
0061     'a'; LATIN SMALL LETTER A
0062     'b'; LATIN SMALL LETTER B
0063     'c'; LATIN SMALL LETTER C
...
007B     '{'; LEFT CURLY BRACKET
...
2167     'Ⅷ'; ROMAN NUMERAL EIGHT
2168     'Ⅸ'; ROMAN NUMERAL NINE
...
265E     '♞'; BLACK CHESS KNIGHT
265F     '♟'; BLACK CHESS PAWN
...
1F600    '😀'; GRINNING FACE
1F609    '😉'; WINKING FACE
...
```

Strings as objects (with methods)

- Python is a object-oriented language
  - All values are objects
- Objects provides methods, i.e., functions that can be invoked onto them, using dotted (or dot) notation

```
object.method()
```

- Thus, for instance, all strings are objects and provide a method named **`upper()`** that returns a new string with all characters capitalized
- Methods are different than “non-object oriented” functions, which in turn are invoked without the dot notation

```
func(param1, param2)
```

Input from the console

- To read a string from the console it is possible to use function **`input()`**

```
name = input("Insert your name")
```

- If a numeric input is required or expected, the string needs to be converted in a number using functions **`int()`** or **`float()`**

```
age_string = input("Insert your age: ") # String
age = int(age_string) # Conversion to int
```

Formatted output

- Sometimes it may help to add numerical values into strings to improve output formatting
- Python offers several alternatives
  - String concatenation using **`+`** operator
  - Formatted-strings, or f-strings
  - Formatting operator **`%`**
  - A method of string named **`.format()`**

Formatted output (comparison)

```
pi = 3.14
r = 2.0
area = (r**2) * pi

print('The area of a circle of radius '+str(r)+' is
      '+str(area))

print(f'The area of a circle of radius {r} is {area}’)

print('The area of a circle of radius %f is %f' % (r, area))

print('The area of a circle of radius {r} is
      {a}'.format(r=r, a=area))
```

F-strings

- A formatted string is a string preceded by `f` or `F`
- These strings can contain replaceable fields, in the form of expressions in `{ }`
- While classical strings typically have a constant value, f-strings are basically expressions evaluated at runtime

```
f"the result is {result}" # the result is 5
f"the sum is {a+b}" # the sum is 15
f"the result is {result=}" # the result is
                                             result=5 (debug)
```

Formatting in f-strings

- It is possible to change the format used to print values, using format specifiers
- Specifiers are included in the `{ }`, after the expression to be evaluated, separated by symbol `:`

```
f"Distance is {dist:8.2} meters”
```

- It is possible to specify alignment (e.g., with `<` for left-alignment), how to handle sign (`+`, `–` or space), the precision (`.` followed by number of digits), the type of conversion (e.g., `s` for string, `d` for integers, `f` for fixed-point numbers), …
  - `f'{pi:10.3f}' # ·····3.142`

Formatting operator

- To control the way in which numbers (e.g., floating point numbers) are printed it is possible to use the `%` operator

```
"string with format specifiers" % (value,value,...)
```

- Each format specifier is replaced, in order, by a value, either calculated or stored in a variable

```
print("Price per liter %.2f" %(price))
# Price per liter: 1.22
print("Price per liter %10.2f" %(price))
# Price per liter: ······1.22
```

Formatting operator

- Main specifiers
  - **`%s`** for strings (or other objects represented as strings, e.g., numbers)
  - **`%d`** (or **`%i`**) for integer numbers
  - **`%f`** for floating point numbers (decimal format)
  - **`%g`** for floating point numbers (exponential format)
  - **`%c`** for single character
- It is apossible to
  - Specify the minimum length of the field, e.g., **`5`**, with **`%5d`**
  - Add heading zeroes instead of spaces, **`%05d`**
  - Indicate the precision (number of decimals) with **`%.<digits>f`**

Formatting operator

- To align a number with two decimals to the right (default)

```
print("%10.2f" %(price))
# ·····17.29
```

- To align a string to the left, symbol – needs to be used

```
print("%-10s" %("Total:"))
# Total:····
```

- To print more data, multiple specifiers can be used

```
print("%-10s%10.2f" %("Total: ", price))
Total:··········17.29
```

Optional parameters of print()

- Function **`print()`** accepts additional, optional parameters
  - **`sep`** defines the separator to use when printing multiple values (default is `' '`)
  - **`end`** defines what to print at the end of a line (default is newline, i.e., **`'\n'`**); it can be used, e.g., to continue printing on the same line
- These are named parameters, meaning that they need to be explicitly named when passing arguments

```
print(hour, min, sec, sep=':')
print('Hello', end='')
```

Decisions

- Building decision with **`if`**, **`else`**, **`elif`**, **`pass`**
- Conditions and operators (relational, boolean, …)
- Nested decisions
- Multiple decisions

Decisions

- In a program it is often necessary to make decisions (or selections, choices) based on the value of a given input, a variable or an expression using some operators (condition)
- Decisions in Python are handled using two main keywords
  - **`if`**
  - **`else`**
- Only one branch is executed
  - The **`if`** one (if condition is **`True`**)
  - The **`else`** one (if condition is **`False`**), when present

Decisions

- The **`if`** and **`else`** instructions are examples of compound statements in Python, which consists of an header (**`if`** or **`else`** keyword, in this case), followed by one or more instructions (block)
- Instructions in the block must have the same indentation
  - Allows the Python interpreter to understand where the block begins/ends, but also supports visual code reading
- Blocks can be nested into other blocks
- Compound instructions require a **`:`** at the end of the header

Decisions

```
##
# This program simulates an elevator that skips the 13th floor
#

# Obtain the floor number from the user as an integer.
floor = int(input("Floor: "))

# Adjust floor if necessary.
if floor > 13:
    actual_floor = floor - 1
else:
    actual_floor = floor

# Print the result
print("The elevator will travel to actual floor", actual_floor)

>>> Floor: 20
>>> The elevator will travel to actual floor 19
```

Conditions and relational operators

- Conditions frequently make use of relational operators
  - `>`, `>=`, `<`, `<=`, `==`, `!=`
- Examples

```
if floor >= 13 :
print("floor is greater than or equal to 13")

if name1 == name2 :
print("strings are identical")

if str1 < str2 :
print("str1 precedes str2 in alphabetical order")
```

Nested decisions

- It is possible to nest an **`if`** compound statement into the brach of another **`if`** compound statement

```
Ask the customer for his/her drink order
if customer orders wine
    Ask customer for ID
    if customer’s age is 21 or over
        Serve wine
    else
        Politely explain the law to the customer
else
    Serve customer a non-alcoholic drink
```

$$\left.\begin{array}{l} \text{if customer’s age is 21 or over} \\ \quad \text{Serve wine} \\ \text{else} \\ \quad \text{Politely explain the law to the customer} \end{array}\right\} \text{Nested } \mathtt{if}$$

Multiple choices

- Necessary to have a way for handling more than two alternatives (branches)
- This way is offered by a statement that complements **`if`**, and **`else`**, that is **`elif`**
- As soon as one of the conditions is verified (**`True`**), the relative block is executed, while remaining conditions are not even checked
- If no condition is verified, the final **`else`** block is executed

Multiple choices

```
if richter >= 8.0 : # Handle the ‘special case’
   first print("Most structures fall")
elif richter >= 7.0 :
   print("Many buildings destroyed")
elif richter >= 6.0 :
   print("Many buildings damaged, some collapse")
elif richter >= 4.5 :
   print("Damage to poorly constructed buildings")
else : # so that the 'general case' can be handled
   last print("No destruction of buildings")
```

Pass

- In some case it may be helpful to write first the overall structure of the decision logic, and fill it in with actual instructions only at a later time
    - However, it is not possible to have a block for an **`if`**, **`else`**, or **`elif`** without instructions in it
    - To deal with this need, a special keywork, **`pass`**, has been defined
    - The **`pass`** instruction does not perform any actual operation but allows to write syntactically correct code where missing instructions will be later replaced by correct ones

Boolean variables and operators

- In Python, **`bool`** is a type for describing boolean values that can be either **`True`** or **`False`**
- Often, boolean variables are used as flags in conditions as they can assume only the above two values (the condition of an **`if`** instruction is boolean)
- Boolean conditions can also be defined using boolean operators: the main ones are **`and`**, **`or`**, **`not`**

```
if 0 <= temp and temp <= 100 :
    print("Liquid")
```

Converting to boolean

- It is possible to convert any value in a boolean using the function **`bool()`**
- This conversion is done automatically when a value is used in the context of an **`if`**, **`while`**, **`and`**, **`or`**, or **`not`**
  - **`if s : # equivalent to if s!='':`**

Strings and boolean functions

- There are number of operators and functions working on strings that leverage boolean values like, e.g., checking whether a string contains a certain substring, verifying whether it starts or ends with a given substring, etc.

```
name = "John Wayne"
if "Way" in name :
   print("The string does contain 'Way’”)

if filename.endswith(".html") :
   print("This is an HTML file.")
```

Other functions for working on strings

- There are other functions that can be used to work on strings, e.g., to validate them (always very important, expecially on data provided as input by the user)

```
count()
find()
isalnum()
isalpha()
isdigit()
isspace()
isupper()
…
```

Validating strings

```
valid = ( len(string) == 13
          and string[0] == "("
          and string[4] == ")"
          and string[8] == "-"
          and string[1:4].isdigit()
          and string[5:8].isdigit()
          and string[9:13].isdigit()
) # validates US phone numbers (###)###-####
```

Iterations

- Cycles
  - **`while`**
  - **`for`** (and **`range`**)
- Nested cycles
- Cycles and strings

Iterations

- Cycles are used to iterate (repeat) several times the same block of instructions (typically with different values of involved variables)
- Python offers two main constructs for handling cycles:
    - `while`
    - `for`

While

- The `while` cycle (i.e., the instructions defined into it) is repeated until a so called "cycle condition" remains `True`
- The condition is
  - Initialized outside the cycle and
  - Updated into it
- If the condition never become `False` (which is a not-so-rare situation), an infinite cycle is produced
  - Infinite cycles should be avoided, as they stuck program execution

While

```
RATE = 5.0
INITIAL_BALANCE = 10000.0
TARGET = 2 * INITIAL_BALANCE

balance = INITIAL_BALANCE
year = 0

while balance < TARGET :
   year = year + 1
   interest = balance * RATE/100
   balance = balance + interest

print("The investment doubled after", year, "years.")
```

Cycles controlled by a counter

- A cycle controlled by a counter
  - Counts the number of repetitions done and
  - Terminates when a predefined number of interations has been reached

```
counter = 1 # Initializes the counter
while counter <= 10 : # Controls the counter
    print(counter)
    counter = counter + 1 # Updates the counter
```

Cycles controlled by an event

- The alternative to a cycle controlled by a counter is cycle controlled by a condition that, sooner or later, becomes **`False`**: in this case, the number of iteration is not known in advance

```
balance = INITIAL_BALANCE # Inits the condition
while balance <= TARGET: # Checks the condition
    year = year + 1
    balance = balance * 2 # Updates the condition
```

Mixed cycles

- Not all the cycles are controlled solely by a counter or an event: in some cases, more complex conditions may be written, using expressions

```
count = 0
ok = True
while (count < 10) and ok : # until count < 10
   a = int(input('Number: '))
   if a==0:                  # and value read !=0
      ok = False
print(f'Number {count+1}={a}') count = count + 1
```

Anticipated termination

- If in a cycle the **`break`** instruction is used
  - The cycle is interrupted without completing current iteration
  - The execution continues with the instructions following the cycle
- If in a cycle the **`continue`** istruction is used
  - The current iteration is not completed
  - The cycle continues normally with the next iteration

For

- The `for` cycle (or `for` ... `in` cycle) is used to iterate on all the values of a container
- A container is an object (like a string) that can store multiple elements
- Python offers various types of containers
  - String, a container of characters
  - List, a container of arbitrary values (numbers, strings, ...)
  - File, a container of text or binary data
  - …

For vs while

```
stateName = "Virginia"
i = 0
while i < len(stateName) :
    letter = stateName[i]
    print(letter)
    i = i + 1
```

```
stateName = "Virginia"
for letter in stateName :
    print(letter)
```

`while`

Index variable **`i`** are assigned values 0, 1, …

`for`

Variable **`letter`** are assigned the elements in the container **`stateName`**

The range container

- The function **`range()`** receives a parameter **`N`** and is used to define a special container hosting a sequence of consecutive numbers, from **`0`** to **`N-1`**
- Hence, a **`for`** cycle using **`range()`** is equivalent to a **`while`** cycle controller by a counter

For with range vs while

```
i = 1
while i < 10 :
    print(i) i = i + 1
```

```
for i in range(1, 10) :
    print(i)
```

Setting the range

- There are several version of **`range()`** function, which allow to define also the starting value or the step

```
for i in range(5) :
  print(i) # 0, 1, 2, 3, 4

for i in range(1, 5) :
  print(i) # 1, 2, 3, 4

for i in range(1, 11, 2) :
  print(i) # 1, 3, 5, 7, 9
```

Different uses of while and for

```
i=0
while i<len(nome):
  # Cycling on indexes
  letter = name[i]
  print(i, letter)
  i = i+1
```

```
for letter in name:
  print(letter)
  # index is not know
  # can be calculated ...

i=0
for letter in name:
   print(letter)
   i=i+1
# ... but is more complex
```

Enumeration

- The function **`enumerate()`** transforms any container in a sequence of pairs, or tuples in the form **`(index, value)`**

```
string = "hello"
enumerate(string) # 0,'h' 1,'e', ...
for (index, value) in enumerate(string) :
    print(f"Letter at index {index} is {value}")
```

Nested cycles

- As much like it is possible to nest **`if`** instructions (using indentation), it is possible to nest **`while`** and **`for`** cycles in order to address complex problems
- A simple example is represented by visiting the cells of a table (or matrix)
  - An (external) cycle can be used, e.g., to iterate on rows
  - An (internal) cycle can then iterate on columns of current row

Nested cycles

```
# This program prints a table of powers of x
# Rows holds values of x, columns the power n, cells x^n
NMAX = 4
XMAX = 10
# Print table header
for n in range(1, NMAX + 1):
    print(f'{n:10}', end="")
print()
for n in range(1, NMAX + 1):
    print(' ' * 8 + 'x ', end="")
print("\n", "    ", "-" * (5 + 10 * (NMAX - 1)))
# Print table body
for x in range(1, XMAX + 1):
    for n in range(1, NMAX + 1):
        print(f"{x ** n:10d}", end="")
    print()
```

Cycles and strings

- Cycles are often used to work with strings
  - To count matches
  - To find all matches
  - To find first and last match
  - To validate a string
  - To build a new string
- Example

```
vowels = 0
for ch in word :
   if ch.lower() in "aeiou" :
      vowels = vowels + 1
```

Lists (basics)

- Creating lists
- Accessing elements and determining the size
- Lists vs strings
- Interating on a list
- Adding elements to a list (**`append`**)

Lists

- A list is a versatile and dynamic data structure which is able to host
  - A variable number of elements,
  - Of any type
- Elements can be accessed based on their position, by index

Creating a list

- A list is created using the operator `[]`
- Once created, the list can be assigned to a variable

```
values = [] # empty list
values = [32, 54, 67, 29, 35, 80, 115]
```

Acessing list elements

- To access list elements, square brackets (i.e., the indexing operator **`[ ]`**) are used together with element position (index), like with strings
- Indexes go from 0 to list length minus 1

```
element = values[1] # accessing a list element
values[i] = 5 # replacing a list element
```

- To determine the size, or length, of a list, the **`len()`** function needs to be used, which returns the number of elements in it

```
num_elements = len(values)
```

Lists vs strings

- Both lists and strings are sequence of elements, the `[ ]` operator can be used to access an element in the sequence while function `len()` can be used to obtain size
- But there are differences between them
  - Lists can contain any type of element, whereas strings only characters
  - Lists are mutable, i.e., value of each element can be updated, new elements can be added, existing ones removed, etc., whereas strings are immutable

Types of list elements

- Lists can contain any type of elements (even mixed, though mixed types shall be avoided, unless there is a specific reason for doing that)

```
short_months = [ 2, 4, 6, 9, 11 ]
math_constants = [ 3.1415, 2.718 ]
short_months2 = [ 'Feb', 'Apr', 'Jun', 'Sep' ]
lucky = [ 'star', 13, 17, '֍' ]
```

Out-of-range errors

- When exceeding list ranges, out-of-range errors are raised
- The most common error is raised when accessing to a non-existing element (non-existing index)

```
values = [2.3, 4.5, 7.2, 1.0, 12.2, 9.0, 15.2]
values[7] = 5.4 # index can go from 0 to 6

>>> IndexError: list index out of range
```

Printing a list with print()

- Using the **`print()`** function, the list is printed in a format similar to that used for creating it
- Individual elements can be printed accessing them with **`[]`**

```
values = [ 1, 2, 3 ]
```

```
>>> print(values)
[1, 2, 3]
>>> print(values[0])
1
```

Iterating on a list using indexes

- To iterate on a list of, say, 10 elements, index values can be used

```
 # First version, length known
for i in range(10) :
    print(i, values[i])

# Second version, using function len()
for i in range(len(values)) :
   print(i, values[i])
```

Iterating on a list using element values

- If the index `i` is not really needed, it is possible to iterate directly on list values, treating the list as a container
- At each iteration, there is a variable that assumes the value of one of the list elements (one at at time)

```
# Third version, without using indexes
for element in values :
    print(element)
```

Using enumeration on a list

- Function **`enumerate()`** can be used to extract from a list all the pairs, or tuples **`(index, value)`**, to later iterate on them

```
# Fourth version, using (indexes, values)
for (i, element) in enumerate(values) :
  print(i, element)
```

Adding elements to a list

- If elements to be added to a list are not know at the time of creating it, an empty list can be created and elements added afterwards using **`append()`**

```
friends = []
friends.append("Harry")
friends.append("Emily")
# lists content: ["Harry", "Emily"]
friends.append("Bob")
# lists content: ["Harry", "Emily", "Bob"]
```

Functions (advanced)

- Functions as black boxes
- Defining functions
- Arguments vs parameters
- Positional and nominal parameters
- Default values for parameters
- Return value(s)
- Function main()
- Naming of functions
- Visibility and scope
- Local and global variables

Functions

- A function is a sequence of instructions to which a name is assigned
  - By factorizing or modularizing code that may be frequently used, optimizations can be made since it is sufficient to call (invoke) the function when those instructions need to be executed
    - An example could be function **`round()`**, which is made of instructions for rounding a floating point number
  - The use of functions, i.e., modularized code, also helps to deal with complex problems to be solved with a computer program by providing a way to decompose them into simpler problems

Functions

- When a function is invoked, it may receive zero, one or more “inputs” (arguments)
  - Can be constants, variables, expressions
- When a function terminates, it may return a result (not necessarily, maximum one), that could be used in an expression (e.g., assigned to a variable)

```
price = round(6.8275, 2) # Assigns 6.83 to price
```

Functions as black boxes

- Functions can be regarded as “black boxes”
- In order to use a function, a programmer does not really need to know its internals
  - Simply pass to the box what it requires to carry out its task, and
  - Get the result
- Only function specifications needs to be known
  - Name
  - Parameters, if any (types, order, and possible default values)
  - Return type, if any

Defining and implementing a function

- To write (define) a function, e.g., for computing the volume of a cube given its edge size
  - A name needs to be chosen (e.g., **`cube_volume`**)
  - Declare a variable for each parameter (e.g., **`side_length`**)
  - Use keyword **`def`** to form the first line of function definition (header)

Defining and implementing a function

- Keyword **`def`** opens a new block (compound statement) where to insert instructions to be factorized
  - The instructions that follow the header, in the block, are called the body of the function

Defining and implementing a function

```
def cube_volume(side_length) :
   volume = side_length ** 3
   return volume
```

Passing parameters

- When a function is called, variables representing parameters (i..e, those in parentheses in the function definition) receive the arguments (“effective” or “actual” parameters) passed with function invocation
  - Variables representing parameters are initialized with the value of the corresponding argument and used as a normal variable in the function
  - When a variable is passed as argument, the variable representing the parameter is initialized with its value: after initialization, two different variables exist
  - The life of the variable representing the parameter will end with the termination of the function (memory is released)

Passing parameters

```
total = 10
new_price = add_tax(total, 7.5)

def add_tax(price, rate):
    # value of total (10) is copied into price
    tax = price * rate / 100
    # No effect outside the function:
    # variable price is not total, though
    # at the beginning have the same value
    price = price + tax
    return price
```

Passing parameters

```
total = 10
new_price = add_tax(total, 7.5)

def add_tax(price, rate):
    tax = price * rate / 100
    # To avoid confusion, it is generally
    # better not to try modifying parameters,
    # simply using different names
    new_price = new_price + tax
    return new_price
```

Commenting functions

- When writing a function, its behavior should be commented (for other humans, not the Python interpreter, which ignores comments)
- There is a standard, named *reStructuredText*, for writing comments (with tools capable of transforming them in ready-to-use documentation)
  - Short description within `"""`
  - List of parameters, with **`:param:`**
  - Return value, with **`:return:`**

Commenting functions

```
def cube_volume(side_length):
    """
    Computes the volume of a cube
    :param side_length: the length of cube side
    :return: the volume of the cube
    """
    volume = side_length ** 3
    return volume
```

Positional and nominal parameters

- In Python, parameters are indicated by the name in the function header
- When invoking the function, arguments can be assigned to parameters in two ways:
  - Using position (default): the first argument will initialize the first parameter, the second argument the second parameter, and so forth
  - Using name: name is used to specify the parameter the argument has to be assigned to

```
def complex(real, imag): ...

x = complex(3, 5)   # 3 + 5j, positional par.
x = complex(real = 3, imag = 5) # 3 + 5j, named par.
```

Default values for parameters

- Parameters can have default values, which are used if, when invoking the function, no argument is assigned to that parameter

```
def complex(real = 0.0, imag = 0.0)

x = complex(0) # 0.0, positional par.
x = complex(imag = 5) # 5j, named par.
x = complex(real = 3) # 3, named par.
```

List of arguments

- The special syntax **`*args`** in a function header is used to indicate that the function can receive a varying number of arguments (without names)
- Such a list is always positional

Return value(s)

- Function can (optionally) return a value using instruction **`return`**, which
  - Immediately terminates the function
  - Passes the return value directly to the function that invoked it (**`main`** or other function)

```
def cubeVolume (sideLength):
    volume = sideLength ** 3
    return volume
```

Return value(s)

- If more than one values need to be returned, a tuple must be constructed (it is still one value, acting as a container)
  - **`return (x, y)`**

Multiple return instructions

- A function may need multiple return instructions, one for each branch of the execution flow

```
def cube_volume(side_length):
    if (side_length < 0):
            return 0
    else:
            return side_length ** 3
```

- A possible alternative is to use a variable to store the return value, which is returned just at the very end of the function

Function not returning a value

- If a function does not need to return a value, it can use the **`return`** instruction without specifying any value

```
return # no value specified
```

- Not using a **`return`** in a function is equivalent to using **`return`** with no value
- Basically, it is like returning the special value **`None`**

The function main()

- When functions are used in Python, it is advisable that all the instructions of a program are included in functions, and that one of these functions is indicated as starting point for program execution
- Any name could be used, in principle, but conventionally the **`main`** name is used for that function
- This function needs to be invoked by an instruction in the program

The function main()

```
def main() :
    result1 = cubeVolume(2)
    result2 = cubeVolume(10)
    print("A cube with side length 2 has volume "
                              + str(result1))
    print("A cube with side length 10 has volume "
                              + str(result2))

def cubeVolume(sideLength) :
    volume = sideLength ** 3
    return volume

main()
```

The function main()

- The **`main()`** function shall be generally invoked only when the program in executed in standalone mode
  - If the program is imported, as a module, in a larger program, the function shall not be invoked
  - To this purpose, it is necessary to check a special variable called **`__name__`**, which contains the name of the module (or the string **`__main__`** if in standalone mode)

```
if __name__ == '__main__':
    # call the function if program is in standalone
    # mode, don't call it has been imported (module)
    main()
```

Naming of functions

- Some Python functions (like variables) have special, reserved names
- To limit confusion, special variables and functions have names that start with __ ("dunder", meaning double-underscore)
- For variables, prefixes _ and __ are also used with other meanings in the context of classes and OO
- For instance, programmers should avoid names for variables and functions that start with _ or __

Using functions

- Functions need to be defined before being invoked (otherwise, an error is raised at runtime)

```
print(cube_volume(10)) # not defined yet

def cube_volume(side_length) :
    volume = side_length ** 3
    return volume


>>> NameError: name 'cube_volume' is not defined
```

- However, a function can be invoked from within another function, e.g., `main()`, even before being defined

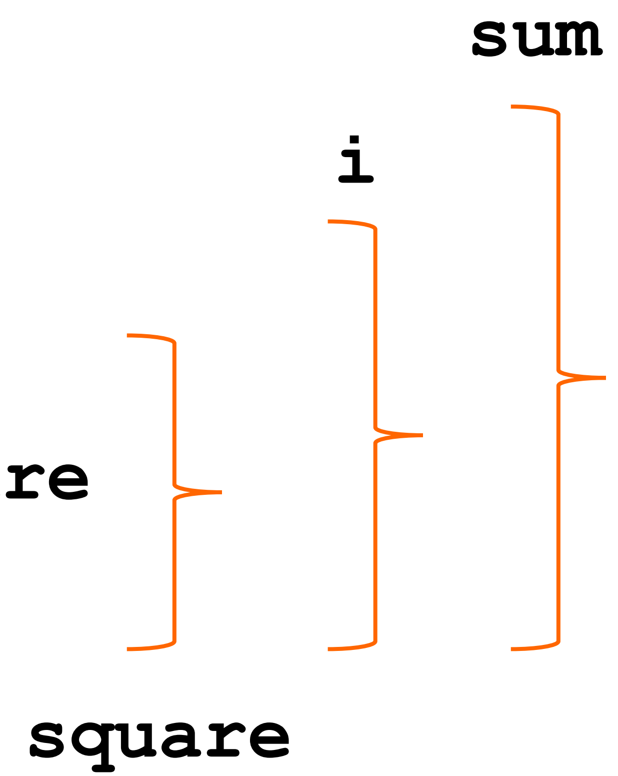
Visibility and scope

- Variables can be defined
  - In a function: they are known ad "local variables", as they are visible, i.e., they are available and can be used only locally, within the function (function parameters are treated as local variables)
  - Outside a function: sometimes they are referred to as "global variables", meaning that they are visible in any function of the program
- The region of the program where the variable is visible and can be used is referred to as scope

Visibility and scope

- Considering variables **`sum`**, **`square`** and **`i`**, each has a different scope

```
def main() :
    sum = 0
        for i in range(11) :
            square = i * i
            sum = sum + square
        print(square, sum)
```



Local variables

- A local variable is a variable defined in a function, which is not visible in another function
- Using it outside the function where it has been defined causes an error at compile time

```
def main():
    side_length = 10
    result = cube_volume()                side_length
    print(result)

def cube_volume():
    return side_length ** 3 # Error, not visible
```

Reusing names for local variables

- Since a variable defined in a function is not visible in another function, this also means that defining another variable with the same name outside the function will not create problems (they would be two different variables)
- It could cause confusion, nevertheless

```
def square(n):
    result = n * n                          }
    return result                           } result

def main():
    result = square(3) + square(4)          }
    print(result)                           } result
```

Global variables

- A global variable is a variable defined outside a function
  - It is visible to all functions
  - But, a function that intends to modify a global variable needs to specify this fact by including a `global` declaration

```
balance = 10000 # A global variable
def withdraw(amount) :
  # Function that updates the global variable
  global balance
  if balance >= amount :
      balance = balance - amount
```

Global variables

- Use (abuse) of global variables could create issues, as a program may update it from any function, and functions cannot be anymore seen as black boxes
- Python natively uses it very rarely, e.g., **`pi`** in **`math`**
- Instead of using global variables, a good (better) practice is to use functions with parameters and return values, which allow to pass from one part of the program to another one in a controlled way

Files

- Reading and writing text files
- File object and its methods
- Working with characters, words, and records/fields
- CSV format

Types of files

- There are basically two types of files:
  - Text files
    - Containing unstructured, free text, organized in terms of rows, words, and characters
    - Containing structured text, made up of records (each row or every a certain number of rows), using separators (CSV, HTML or custom)
  - Binary files
    - Containing various type of multimedia data

Opening a file for reading

- To access a file it is first necessary to open it
- If the intention is to read the file, function **`open()`** needs to be used passing as arguments the name of the file and the string **`"r"`**
- The function returns a file object that will be used to access the content (hence, it is typically stored in a variable, on which methods are invoked using dot notation)
- File needs to exists, otherwise an error is raised

```
infile = open("input.txt", "r")
```

Opening a file for writing

- To open a file for writing, function **`open()`** needs to be used, like for reading, but passing as argument, besides the name of the file, the string **`"w"`**
- The function returns a file object that will be used to operate on it
- If the file does not exist, a new, empty file is created
- If the file already exists, its content is cleaned (the alternative is to use **`"a"`** in place of **`"w"`**, for appending)

```
outfile = open("output.txt", "w")
```

Closing a file

- After having operated on the file (reading or writing), it is always important to close it invoking method **`close()`** on the file object

```
infile.close()
outfile.close()
```

- Should a program terminate with an open file on which writing operations were in progress, the output may not be fully saved into it

Closing a file

- In order to help the programmer avoid forgetting to close a file, Python offers a special shortcut, based on the use of keyword **`with`**

```
with open(filename, "w") as outfile :
   … # e.g., write output or perform other ops
```

- The instruction opens the file and assigns the corresponding object to the indicated variable, automatically closing it once all the instructions in the block have been executed (of when an error is raised)

Locating files

- The file is searched for into the folder where the Python program is executed
  - When an IDE is used, like PyCharm, this is the main folder of the open project
  - It is generally possible to configure the IDE in a different way, so that it executes the program from the file directory (a further configuration may be needed for debug mode)

UTF-8

- If a text file contains special characters not included in basic ASCII coding, it is typically encoded in Unicode UTF-8
  - https://docs.python.org/3.7/library/functions.html#open
- Function **`open()`** uses the default encoding of the operating system
  - To specify that the file shall be opened with a particular encoding, arguments **`encoding`** has to be added when invoking function **`open()`**

```
infile = open('file.txt','r', encoding='utf-8')
```

Reading a file

- When a file is opened, an imaginary “cursor” is positioned at the beginning of file itself
- Several methods are available to read a part of the file, starting from cursor position (which is moved accordingly after the reading operation)

  `infile.read(1)` to read one character

  `infile.read(N)` to read **`N`** consecutive characters

  `infile.readline()` to read a line, with all characters in it

  `infile.readlines()` to read all lines

- When reading lines, end-of-line is reached and read

Reading a whole file

- When the end of the file has been reached, the reading operation returns an empty string, i.e., `""`
- To read a file, a cycle can be used and not having reached end-of-file can be used as cycle condition

```
line = infile.readline()
while line != "" :
   line = infile.readline() # Read a line
   # Process the line just read ...
```

Reading a whole file

- Python can also treat an open file as a container of strings, with a kind of implicit call to **`readline()`**
- At each interaction, the variable used in the **`for`** assumes the value of the next line

```
for line in infile :
  print(line)
```

- Files are special types of containers, though: once read, they need to be closed and reopened to be read again

Reading a whole file

- A further method to read a whole file in a single operation is to invoke method **`read()`** with no argument, which will return a string with the entire file content
  - **`contents = infile.read()`**

Processing text read from files

- Data are read from a text file as strings: hence, if there are data that need to be interpreted as numbers, conversion functions shall be used
  - End-of-line is ignored by functions like **`int()`** and **`float()`**
- Nevertheless, strings read from the file contain the end-of-line: to remove it, method **`rstrip()`** can be used, operating from the right-end of the string

```
line = line.rstrip('') # rem. '\n' and trailing spaces
line = line.rstrip('\n') # rem. only the '\n'
```

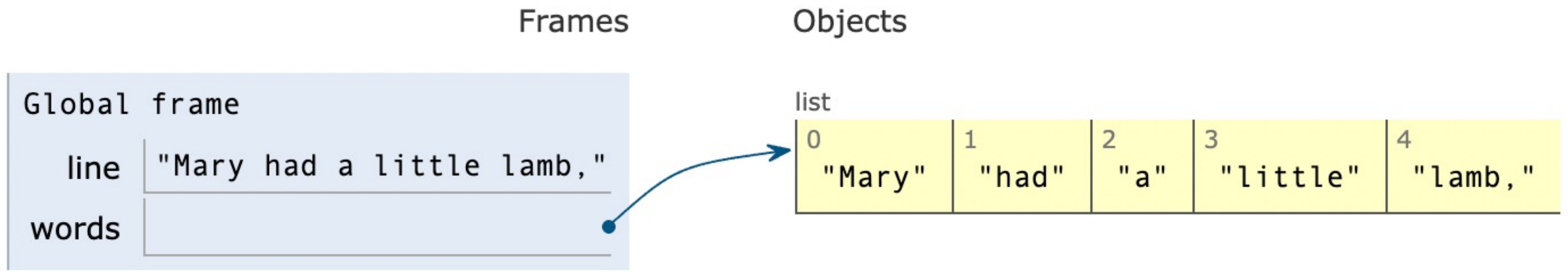
Reading words from files

- Sometimes, words need to be read: since, there is not a dedicated method to read words, strings have to be read first, then “tokenized” using a method like **`split()`**

```
line = 'Mary had a little lamb,'
words = line.split()
```

- To tokenize entire lines, rather than words (i.e., extracting all the single lines from a string), method **`splitlines()`** can be used instead
  - **`lines = contents.splitlines()`**

Reading words from files



Reading words from files

```
"""
Open a text file, then iterate on all its lines
Split each line read in words removing tailing
characters, and print them all on separate lines
"""

input_file = open("lyrics.txt", "r")
for line in input_file :
     word_list = line.split()
     for word in word_list :
          word = word.rstrip(".,?!")
          print(word)
input_file.close()
```

Reading a file one character at a time

```
char = input_file.read(1)
while char != "" :
    # Process the character read
    char = input_file.read(1)
```

Writing on a file

- To write on a file that has been opened for writing it is necessary to use function **`write()`**
- Differently than with **`print()`**, when writing on a file it is necessary to explicity add a **`"\n"`** for moving to next line

```
outfile.write("Hello, World!\n")
```

- It may be helpful to write on files using f-strings

```
outfile.write(f"Number of entries: {count}\n
                 Total: {total:8.2f}\n")
```

Writing on a file

- It is also possible to write on a file multiple lines at once using function **`writelines()`**, which receives as argument a list of strings (each string needs to be already terminated with **`"\n"`**, if needed, as it is not added automatically)
- Also the **`print()`** function can be used to write on a file, by redirecting its output with the optional parameter **`file`**

```
print("The result is:", val, file=outfile)
```

Working with CSV files

- CSV is the acronym for Comma Separated Values, a common format for exchanging structured (tabular) data
- Records (table rows) are reported on different lines, whose fields are separated using commas ','
- To let fields include commas as characters, " " are used

```
"Detective Story", "1951", "William Wyler"
"Airport 1975","1974","Jack Smight"
"Hamlet","1996","Kenneth Branagh"
"American Beauty","1999","Sam Mendes"
"Bitter Moon","1992","Roman Polanski"
...
```

Working with CSV files

- A CSV file can be processed, e.g., by
  - Reading lines, one at a time with **`readline()`** or all at once, with **`read()`**, then splitting them with **`splitlines()`**
  - Processing each single line (record) to obtain fields, using **`split(',')`**
  - Since fields may be enclosed in quotes, **`strip('"')`** could have to be used to remove them

Working with CSV files

- Alternatively (better), the **`csv`** module of the standard library (and its functions) can be used
- For instance, function **`reader()`** receives as input a file open for reading and returns an object that can be iterated
  - Each iteration returns a list (corresponding to a record) of strings (its fields)

```
from csv import reader
csvReader = reader(infile)
for row in csvReader :
   print(row)
```

Working with CSV files

- In a comparable way, function **`writer()`** can be used for writing a CSV file given a file object open for writing

```
from csv import writer
csvWriter = writer(outfile)
csvWriter.writerow(["J. Smith",1607,"Senior",3.28])
... # write other rows
```

Exceptions

- Detecting errors (anomalies)
  - **`try`**
- Handling them
  - **`except`**, **`finally`**
- Firing them
  - **`raise`**

Why exceptions

- When functions are used, it may happen that, in a certain part (function) of a program, an error is raised which should be managed, i.e., “recovered”, elsewhere
  - For instance, in **`main()`**, a function **`requestFilenames()`** may be called that asks the user to provide two filenames; then another function **`readData()`** is called that, in turn, calls two functions, **`readFirstFile()`** and **`readSecondFile()`**, each devoted to reading one of the two files: if, e.g., the latter function fails because the file does not exists, the control should be passed back to the function that requests the filenames

The mechanism of exceptions

- Exceptions are a flexible mechanism introduced in programming languages to improve error management, by separating error detection from error handling
- With exceptions, the control is passed from the piece of code which detected the error to the piece of code that can handle it

Exception detection (catching)

- Detecting an error
  - Means realizing, at run-time, that there is a problem (anomaly) which prevents normal continuation of the program
  - Typically in a function, which may have been invoked in other functions (possibly from a library)
  - Function does not know how to “recover” from the error and needs to inform another part of the program which knows that

Exception handling

- Handling an error
  - Means receiving the information about that fact that an error has been detected (in/from an invoked function)
  - Understand what is the error, and which are its causes (critical operation)
  - Try to resolve the (causes of the) error and possibly retry the operation
  - In the worst case, interrupt program execution

Implementing the mechanism

- The mechanism for detecting and handling exception is implemented with instructions **`raise`**, **`try`** and **`except`**
  - With **`raise`**, the function must check that all the conditions for normal execution are met, otherwise it raises an exception
    - There are many types of exceptions, depending on the cause (**`ValueError`**, **`OSError`**, ...), and each exception can be assigned a message describing the problem
  - If functions that can raise exceptions are used, a special **`try`** ... **`except`** code for handling them must be used
    - Code to be controlled is included in a **`try`** block (for catching exception)
    - Code for handling the exception is included in an **`except`** block
    - If the exception is not handled, the program will be interrupted

Implementing the mechanism

- Instructions that could raise exceptions should be included in a **`try`** block, and one or more **`except`** clauses should be used to provide the handler for the particular exception that may be raised in the try block

```
try :
    filename = input("Enter filename: ")
    infile = open(filename, "r") # could generate an OSError
    line = infile.readline()
    value = int(line) # could generate a ValueError
    ...
except OSError :
    print("Error: file not found.")
except ValueError as exception :
    print("Error:", str(exception))
```

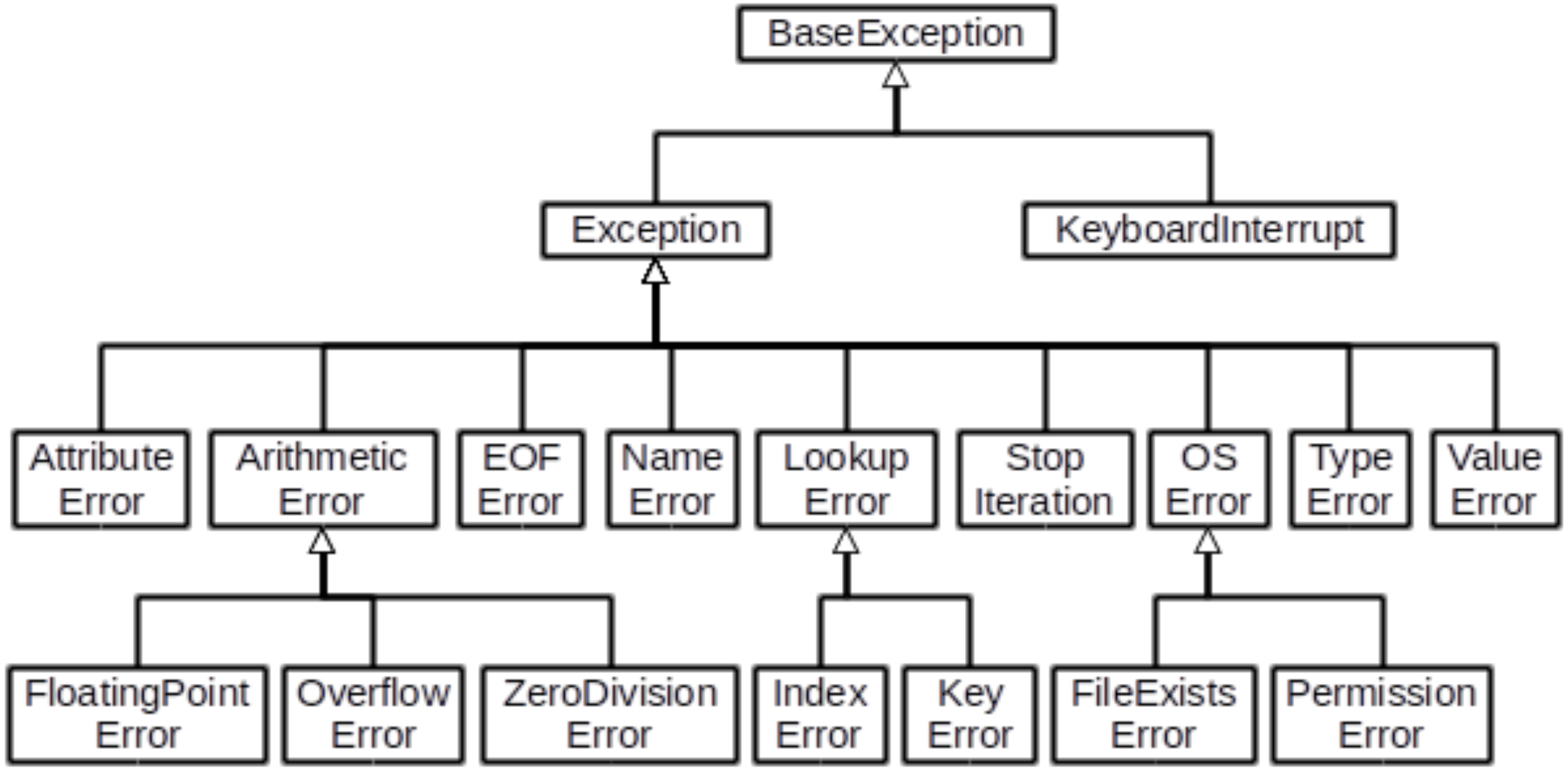
Implementing the mechanism

- All the exceptions should be handled somewhere in the program
  - It is something quite complicated to do that
  - It would require to handle any possible type of exception, reacting to them in the proper way
  - Not all the errors can necessarily be recovered
- In some cases (e.g., for errors that cannot be recovered)
  - It may be easier to terminate the program
  - Or, for improved usability, asking the user to fix the error

Examples of exceptions

```
val/0
>>> ZeroDivisionError: division by zero
int('goofy')
>>> ValueError: invalid literal for int() w/ base 10: 'goofy'
a
>>> NameError: name 'a' is not defined
l[10]
IndexError: list index out of range
d['goofy']
KeyError: 'goofy'
open('goofy.txt')
FileNotFoundError: [Errno 2] No such file or dir: 'goofy.txt'
```

Main exceptions of the standard library



Try … except

- If an exception is raised in the **`try`** block, all the following instructions in the block are skipped, and control passes to the **`except`** block for that exception, if defined
- Once the instructions in the **`except`** block have been executed, control is returned after the **`try … except`** block
- If other exceptions are raised in **`try`** block for which an **`except`** block is not defined, program execution will be interrupted
- To catch all exceptions it is possible to
  - Use a single **`except`** clause without specifying the exception type
  - Use **`except Exception`**, where **`Exception`** is a generic exception type encompassing all the other types

Exception object

- To get the message contained in an exception, the exception object needs to be accessed
- The object can be obtained using the keyword **`as`**
- When the handler is executed, the specified variable will be set to the exception object generated by the **`raise`**
- The message can be obtained by converting the object into a string with **`str()`**

```
except ValueError as exception :
    print("Error:", str(exception))
```

Finally

- There is further keyword, **`finally`**, used to executed some “final” operations after the **`except`** blocks, regardless of the fact that exceptions have actually been raised
- An example is closing an (output) file even in case of exceptions (to ensure that file content generated till that point has been written)

```
outfile = open(filename, "w")
try :
    writeData(outfile)
finally :
    outfile.close()
```

Raise

- Many times, exceptions are raised by functions in other libraries
- In some cases, it is the programmer that wants to signal an error condition using the same mechanism, i.e., raising an exception (possibly accompanied by a message describing it)
- When the execution is raised, control is passed to the handler (that catches it, if defined)

```
if amount > balance :
    raise ValueError("Amount exceeds balance")
```

Exceptions and files

```
done = False
while not done :
   try:
      # Prompt user for file name
      data = read_file(filename) # May raise exceptions
      # Process data
      done = True
   except OSError:
      print("File not found.")
   except ValueError :
      print("File contents invalid.")
   except RuntimeError as error:
      print("Error:", str(error))
```

Exceptions and files

```
def read_file(filename) :
   inFile = open(filename, "r") # May throw exceptions
   try:
      return read_data(inFile)
   finally:
      inFile.close()
```

Exceptions and files

```
def read_data(in_file) :
   line = in_file.readline()
   number_of_values = int(line) # May raise a ValueError
   data = []
   for i in range(number_of_values) :
      line = in_file.readline()
      value = float(line) # May raise a ValueError
      data.append(value)
   # Make sure there are no more values in the file
   line = in_file.readline()
   # Extra data in file
   if line != "" :
      raise RuntimeError("End of file expected.")
   return data
```

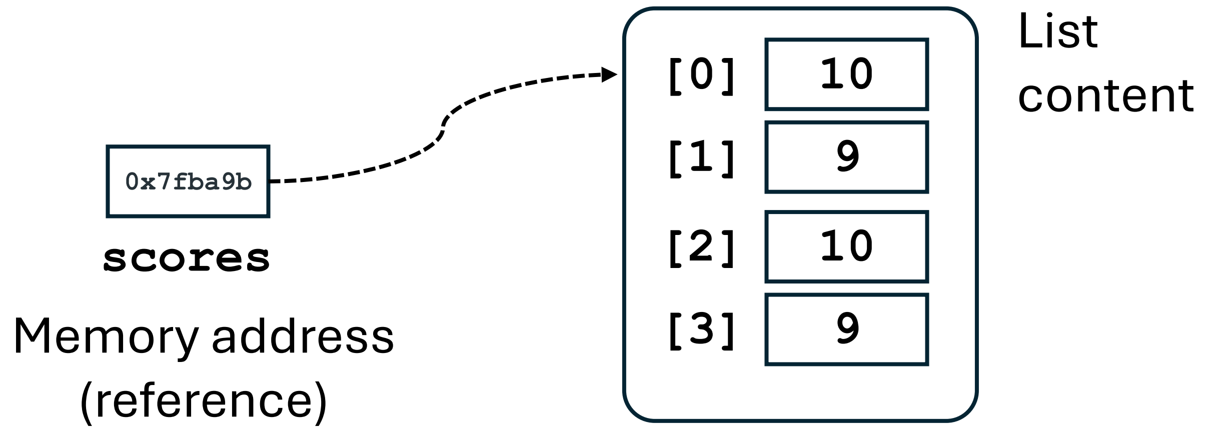
Lists (advanced) and other data structures

- References to lists and aliases
- Other operations on lists (**`insert`**, **`in`**, **`index`**, **`pop`**, **`remove`**, **`extend`**, …)
- Copying, sorting, and slicing lists
- Tuples, tables, sets, and dictionaries
- Advanced sorting

References to lists

- There is an important difference between
  - The name of the list, i.e., the name of the variable that is used to make reference to the list
  - The actual list, i.e., the memory area where its content, i.e., the elements, are actually stored
- The variable contains the reference, i.e., the "pointer" to that memory area

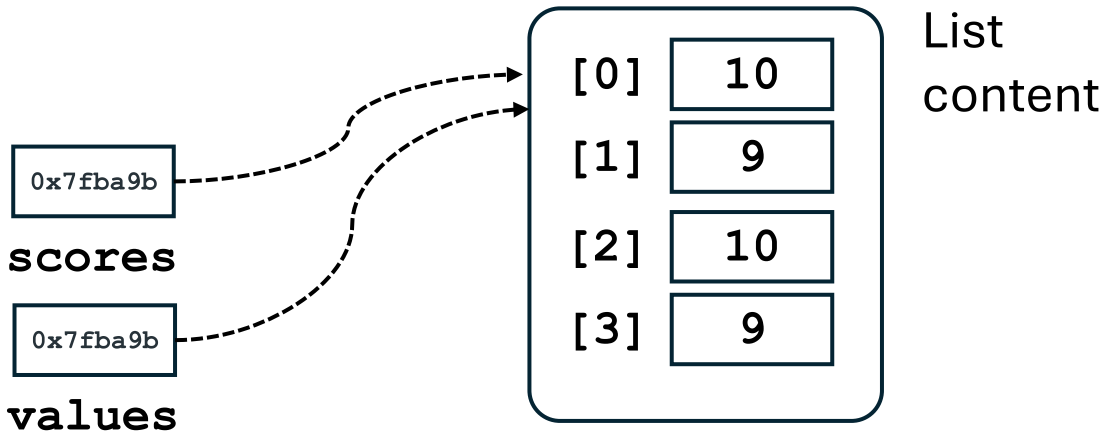
```
scores = [10, 9, 7 , 4]
```



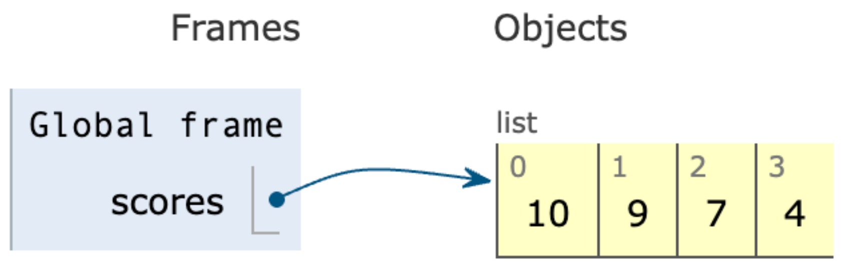
Aliases of lists

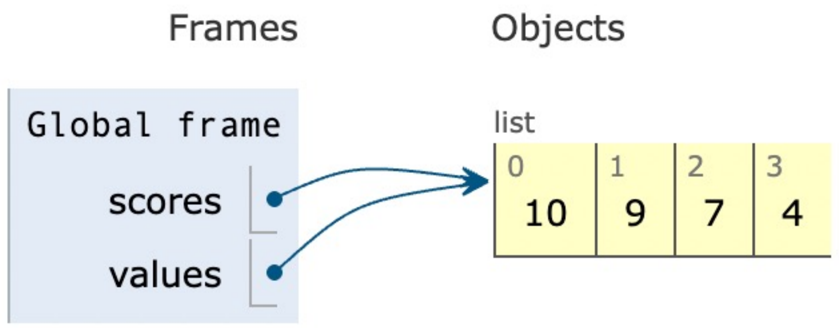
- The effect of an instruction which copies (the value of) a variable pointing to a list into another variable is that, once the instruction has been executed, there will be two variables pointing to the same list
  - Aliasing: the second variable is an alias of the first one
  - There is only one list in memory (content is not copied)

```
scores = [10, 9, 7 , 4]
values = scores
```



Aliases of lists



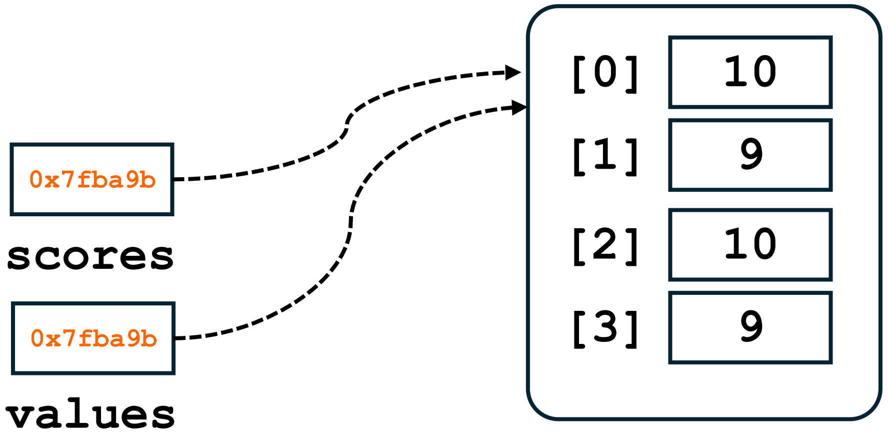


Working with lists using aliases

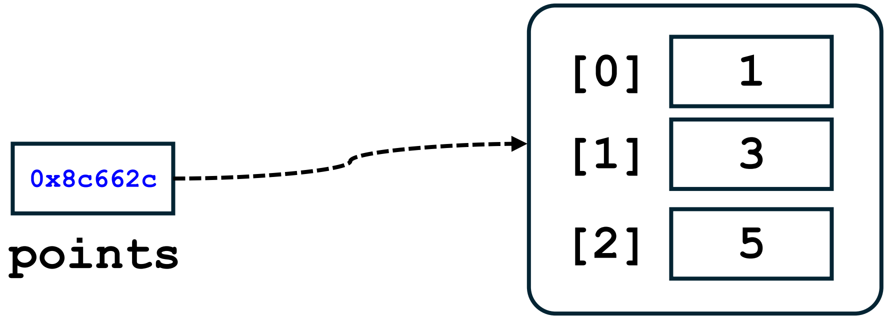
- It is possible to access (also for modifying) the content of a list using either either name/alias
- A list can have one or more names/aliases
- There are situations in which a list has no more name/alias pointing to it
  - It is still in memory but is not reachable anymore, there is no reference to it (“zombie”)
  - Sooner or later, a mechanism called “garbage collector” will free unused memory

Working with lists using aliases

```
scores = [10, 9, 7 , 4]
values = scores
```



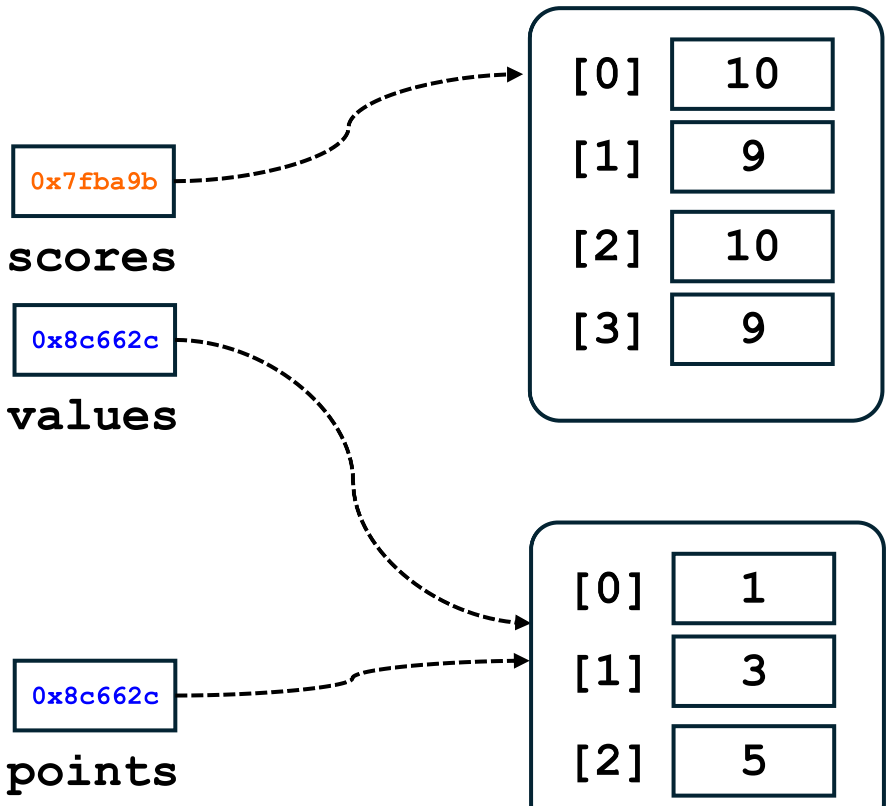
```
points = [1, 3, 5]
```



Working with lists using aliases

```
scores = [10, 9, 7 , 4]
values = scores
```

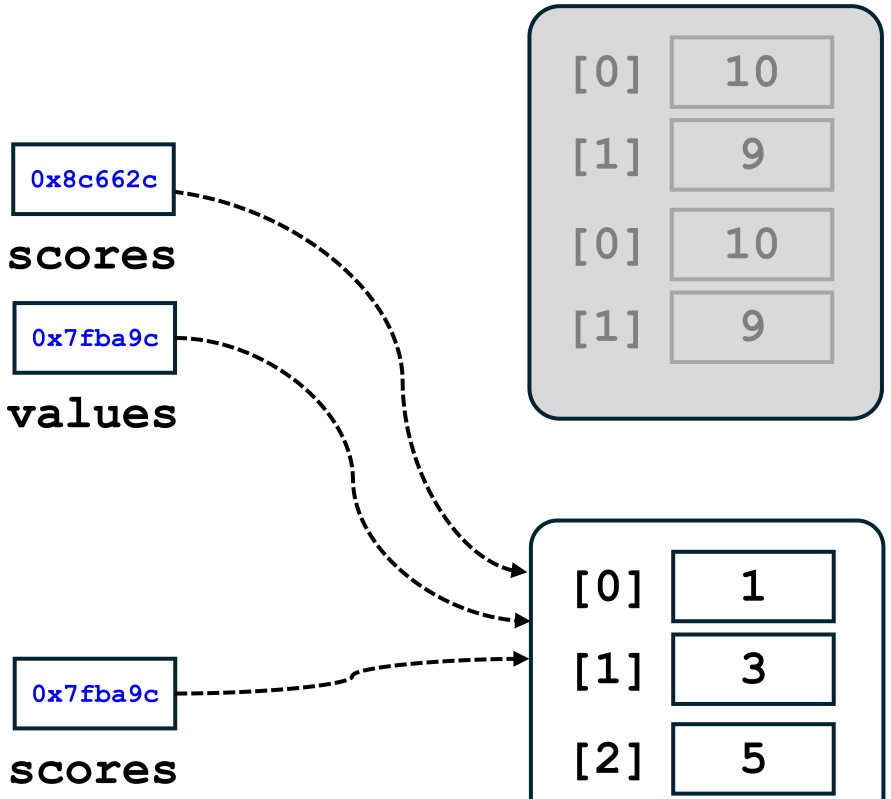
```
points = [1, 3, 5]
values = points
```



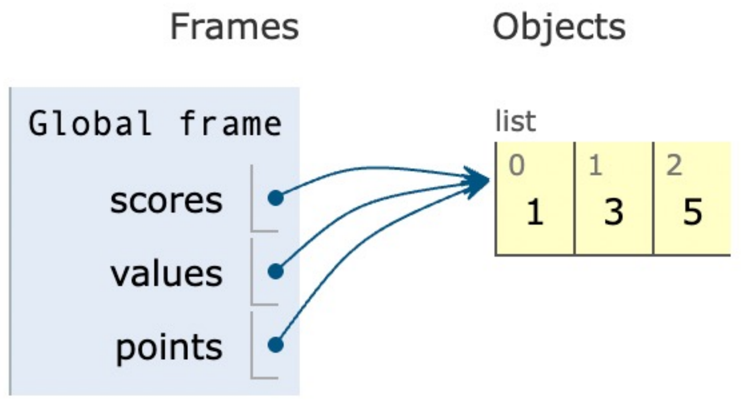
Working with lists using aliases

```
scores = [10, 9, 7 , 4]
values = scores
```

```
points = [1, 3, 5]
values = points
scores = points
```



Working with lists using aliases

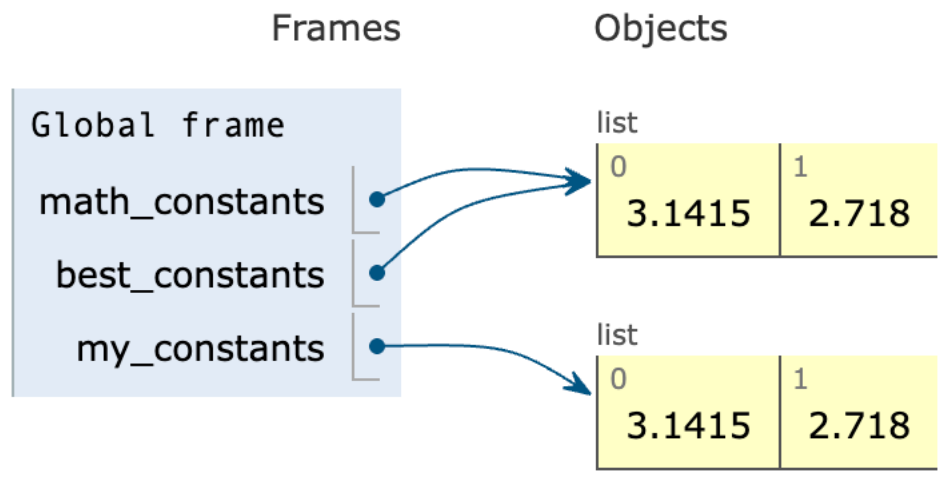


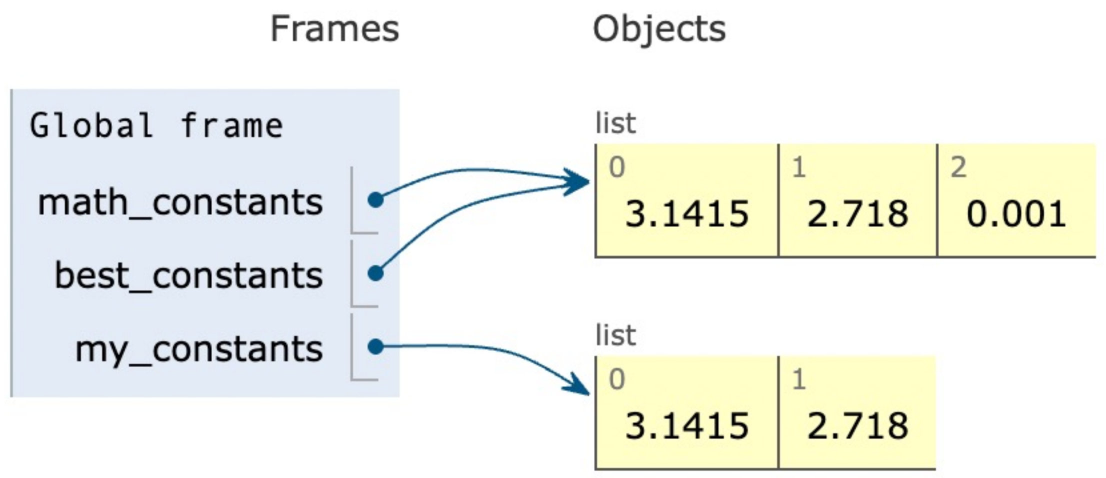
Creating a copy of a list

- To create a copy of a list (i.e., create a copy of its content), a new list needs to be created with same elements in the same order of the original list
- To this purpose, function `list()` shall be used

```
math_constants = [ 3.1415, 2.718 ]
best_constants = math_constants
my_constants = list(math_constants)
math_constants.append(0.001)
```

Creating a copy of a list





Negative indexes

- Differently than other languages, Python allows the use of negative indexes to access list elements in reverse order
  - For instance, `[-1]` gives access to last element in the list, `[-2]` to the element before the last one, etc.

Operations on lists

- There are a number of operations that can be performed on lists besides creating and adding elements to them:
  - Inserting elements
  - Finding elements
  - Removing an element
  - Concatenating, extending, and replicating lists
  - Performing equality/inequality tests on lists
  - Calculating the sum of elements
  - Finding min and max element
  - Sorting elements
  - Slicing lists

Inserting elements

- With **`append()`**, the new element is added after existing ones
- In some cases, the exact insertion position needs to be specified: in this case, **`insert()`** shall be used, which makes space for the new elements, moving "forward" all the already existing ones

```
friends = ["Harry", "Emily"]
friends.insert(1, "Cindy")
# list content: ["Harry", "Cindy", "Emily"]
```

Checking for an element presence

- To determine whether an element is present or not in a list it is necessary to use the operator **`in`**
- The result is a boolean value:
  - **`True`**, if element is present in the list
  - **`False`**, if not present

```
if "Cindy" in friends :
   print("She's a friend")
```

Searching for an element (index)

- It is often helpful to know the position of an element in a list
- In this case, **`index()`** provides the index of the first occurrence
  - It returns an integer value if the element is present
  - Otherwise, if not present, it raises a **`ValueError`**

```
friends = ["Harry", "Emily", "Bob", "Emily"]
n = friends.index("Emily") # set n a 1
```

Removing an element

- With **`pop()`** it is possible to remove the element at the position passed as parameter
- All the elements following the removed one are moved “backward” and list length is reduced by one

```
friends = ["Harry", "Cindy", "Emily", "Bob"]
friends.pop(1)
# lists content: ["Harry", "Emily", "Bob"]
```

Concatenating lists

- The concatenation of two lists, using the **+** operator, creates a new list that contains the elements of the first list followed by the elements of the second list

```
myFriends = ["Fritz", "Cindy"]
yourFriends = ["Lee", "Pat"]
ourFriends = myFriends + yourFriends
# ourFriends: ["Fritz", "Cindy", "Lee", "Pat"]
```

Extending lists

- The **+** operator creates a new list: if an existing list needs to be extended appeding the content of another list to it, **`extend()`** needs to be used

```
evenNumbers = [2, 4, 6, 8]
oddNumbers = [1, 3, 5]
evenNumbers.extend(oddNumbers)
# evenNumbers : [2, 4, 6, 8, 1, 3, 5]
```

Replicating lists

- List replication, with **`*`** operator, generates multiple copies of its elements, like with strings

```
monthInQuarter = [ 1, 2, 3 ] * 4
# [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2 ,3]
```

- Replication is often used to initialize a list with a fixed value

```
monthlyScores = [0] * 12
```

Performing equality/inequality tests

- To check whether two lists have the same elements in the same order, the **==** operator can be used

```
[1, 4, 9] == [1, 4, 9] # True
[1, 4, 9] == [4, 1, 9] # False
```

- The contrary is obtained using **!=**

```
[1, 4, 9] != [4, 9] # True
```

Calculating the sum, finding min/max

- On a list containing number values, **`sum()`** calculates the sum of all elements
  - `sum([1, 4, 9, 16]) # returns 30`
- On a list of number or strings, **`min()`** and **`max()`** return the minimum and maximum element, respectively

```
min(["Fred", "Ann", "Sue"]) # returns "Ann"
max([1, 16, 9, 4]) # returns 16
```

Sorting a list

- With **`sort()`** it is possible to sort a list of numbers or strings in ascending order (default)
  - A new list is not created: rather, after the execution of the method, the original list is actually sorted
  - To create a new list, **`sorted()`** shall be used instead

```
values = [1, 16, 9, 4]
values.sort()              # values: [1, 4 , 9, 16]

values = [1, 16, 9, 4]
other = sorted(values)     # other: [1, 4 , 9, 16]
                           # values unchanged
```

Sorting a list (reverse order)

- To sort a list in reverse (i.e., descending) order, the optional **`reverse`** parameter of **`sort()`** and **`sorted()`** shall be used

```
values = [1, 16, 9, 4]
values.sort(reverse=True) # values: [16, 9, 4, 1]
```

Slicing lists

- The slicing mechanism working on strings can be used in the same way also on lists, using operator `:`
- Arguments are first element (index `0`, if omitted), last element (not included, if omitted end of the list), plus optional step (if negative, reverse order)

```
# temperatures over 12 months
temps = [18, 21, 24, 33, 39, 40, 39, 36, 30, 22, 18]
# temperatures of Q3 (indexes 6, 7, 8)
thirdQuarter = temps[6 : 9]
```

Other methods on strings using lists

- Methods **`join()`** combines elements using a string as separator

```
new_names = ', '.join(names)
```

- Method **`split()`** returns a list substrings (tokens) resulting from the splitting (“tokenization”) of a string at each separator character (space, tab, or newline), possibly provided as parameter

```
wordList = line.split()
wordList = line.split(",")
```

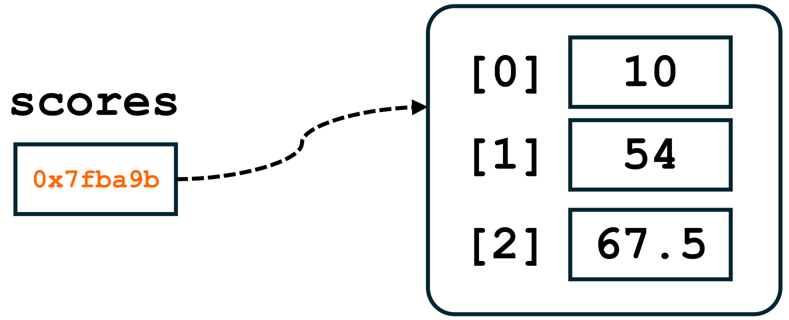
Passing lists to functions

- A function can accept a list as argument (or more than one)
  - The list is accessible within the function though its name (a reference, or pointer to the memory area where list elements are stored)
  - If the function modifies the list, it directly changes the content of that memory area (hence, other functions with variables making reference to the list will see it modified too)

```
scores = [32, 54, 67.5]
def multiply(values, factor) :
for i in range(len(values)) :
  values[i] = values[i] * factor
```

Passing lists to functions

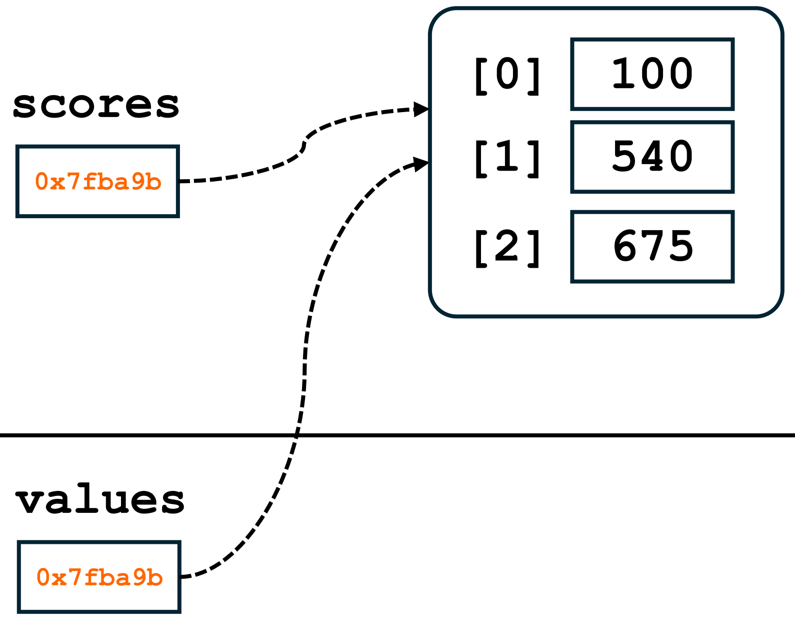
```
scores = [32, 54, 67.5]
```



Passing lists to functions

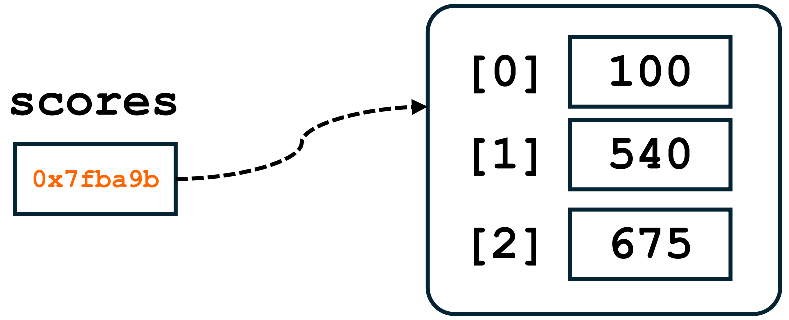
```
scores = [32, 54, 67.5]
multiply(scores, 10)
```

```
def multiply(values, factor) :
for i in range(len(values)) :
      values[i] = values[i] * factor
```



Passing lists to functions

```
scores = [32, 54, 67.5]
multiply(scores, 10)
```



Returning lists from functions

- To return a list it is sufficient to use the **`return`** instruction followed by a list value

```
def squares(n) :
    result = [] # prepares the list to be returned
    for i in range(n) :
        result.append(i * i)
    return result
```

Tuples

- A tuple is simular to a list, but once created its content cannot be modified (it is an immutable version of a list)
  - It is an ordered and immutable container
  - Used, e.g., to return multiple values from a function, with **`enumerate()`** to iterate on a list, etc.
- A tuple is created by specifying its content as a sequence of elements in **`( )`**
- Parentheses may be omitted, but it is not recommended
  - **`vertex = (5, 10, 15)`**

Operations on tuples

- Although tuples are created using **`( )`**, they are accessed like lists using **`[ ]`**
- Tuples can be concatenated using operator **`+`**
- Functions **`len()`**, **`sum()`**, **`max()`**, … work also on tuples
- Is it possible to iterate on tuple elements using **`for`**
- Operators **`in`** and **`not in`** can be used also to check for element existence in tuples

Assigning values to tuples

- In Python, it is possible to assign values to multiple variable in a single instruction using tuples

```
(h, m, s) = (12, 32, 55)
```

- Tuples are also helpful to exchange variables

```
values = [10, 20, 30]
(values[0], values[2]) = (values[2], values[0])
```

Returning values from functions

- Tuples are used to return multiple values from functions

```
def read_date() :
   print("Enter a date:")
   month = int(input(" month: "))
   day = int(input(" day: "))
   year = int(input(" year: "))
   return (month, day, year) # returns a tuple

# calling the function
date = read_date()

# calling the function extracting values from tuple
(month, day, year) = read_date()
```

Comprehension

- Often it is necessary create a list (or tuple) starting from another data structure (list, tuple, string, range) performing a given operation on all elements
- An approach could be to use method **`append()`** in a cycle

```
days = ['Monday','Tuesday','Wednesday’,
        'Thursday’, 'Friday','Saturday’,
        'Sunday']
lengths = []
for day in days :
    lengths.append(len(day))
print(lengths) # [6, 7, 9, 8, 6, 8, 6]
```

Comprehension

- Syntax is clear, but verbose: it can be simplified with an approach that, for lists, goes under the name of list comprehension (meaning inclusion): a new list is created with lists from another container, possibly after modification

```
lengths = [len(day) for day in days]
```

- A condition can be used to select only elements matching it

```
lengths = [len(day) for day in days if
           day.startswith('T')] # [7, 8]
```

- Also works with tuples, sets and dictionaries

Tables

- Lists can be used also to host data structured along two dimensions, like those of a spreadsheet (table, matrix)
- Two dimensional matrices, can be created nesting the `[ ]` operator: the effects is the creation of a list of lists

```
COUNTRIES = 5
MEDALS = 3
Counts = [
        [ 0, 3, 0],
        [ 0, 0, 1],
        [ 0, 0, 0],
        [ 1, 0, 0],
        [ 0, 0, 1]
]
```

Tables

- Sometimes such a list needs to be created incrementally
- To this aim
  - First the list that will host the rows is created
  - Then, a new list is created for each row (representing the columns)

```
table = []
ROWS = 5
COLUMNS = 20
for i in range(ROWS) :
   row = [0] * COLUMNS
   table.append(row)
```

Accessing elements of a table

- To access the elements of a table, two indexes are used: first the index for the row, then the index for the column (both starting from `0`, as for lists)

```
medalCount = counts[3][1] # row 3, column 1
```

Accessing elements of a table

- To iterate on all the elements of a table, nested cycles can be used (external one on rows, internal one on columns)

```
for i in range(COUNTRIES):
    # processing i-th row
    for j in range(MEDALS) :
        # processing j-th column of i-th row
        print("%8d" % counts[i][j], end="")
    print() # adding a newline at end of the row
```

Using tables with functions

- When passing a table, e.g., named **`values`** to a function, it is necessary to extract its dimensions to work on it in the function
  - **`len(values)`** is the number of rows
  - **`len(values[0])`** is the number of columns

```
def sum(values) :
    total = 0
    for i in range(len(values)) :
        for j in range(len(values[0])) :
            total = total + values[i][j]
    return total
```

Sorting tables

- It is possible to use **`sort()`** or **`sorted()`** on a table
  - Sorting will compare (and possibly exchange) entire rows
  - Single rows are compared as for lists

```
a=[
   [3, 1, 4],
   [1, 5, 9],
   [2, 6, 5],
   [3, 5, 8],
   [9, 7, 9]
]
```

```
[
 [1, 5, 9],
 [2, 6, 5],
 [3, 1, 4],
 [3, 5, 8],
 [9, 7, 9]
]
```

Copying tables

- When tables, which are made up of lists containing other lists, need to be “copied”, there are several approaches that can be considered, with different outcomes
  - Creating an alias: new variable, no data is actually copied

    ```
    alias = table
    ```

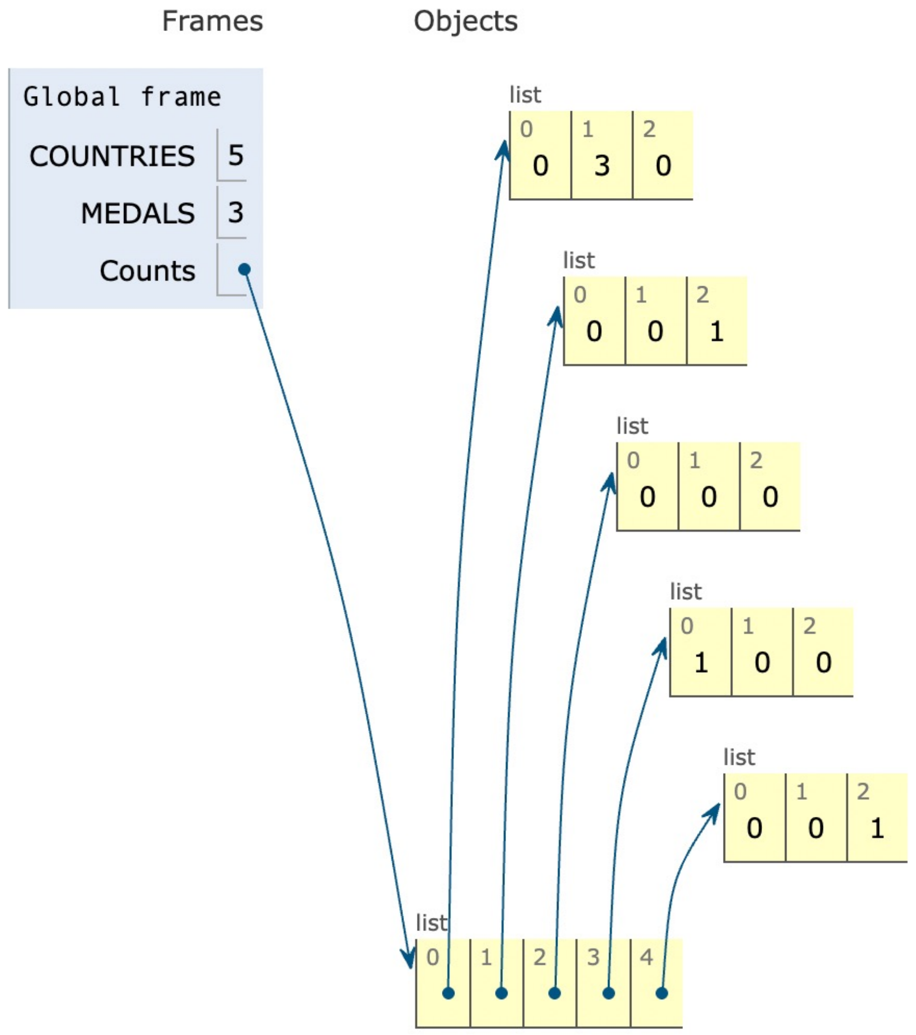
  - Shallow copy: copy the list, without copying the elements (shared between the two copies)

    ```
    shallow = list(table)
    ```

  - Deep copy: copy the whole list, replicating all its elements

    ```
    import copy
    deep = copy.deepcopy(table)
    ```

Tables



Frames
Objects
Global frame
COUNTRIES 5
MEDALS 3
Counts
list
0 1 2
0 3 0
list
0 1 2
0 0 1
list
0 1 2
0 0 0
list
0 1 2
1 0 0
list
0 1 2
0 0 1
list
0 1 2 3 4

Sets

- A set is another type of container storing a collection of unique values (by definition, duplicates are not allowed)
- Differently than with lists, elements
  - Are not recorded in a particular order
  - Cannot be accessed by position
- Operations are those defined on mathematical sets
- Since they don't have to keep a particular order, operations performed on sets are faster than on lists
- Values need to be immutable and comparable (numbers, strings, tuples, objects, but not lists, dictionaries, files, sets)

Defining a set

- Sets can be created using `{ }`

```
cast = { "Luigi", "Gumbys", "Spiny" }
```

- Alternatively, it is possible to use function `set()` to convert any sequence of elements (string, list, tuple, range, set) into a set

```
names = ["Luigi", "Gumbys", "Spiny"]
cast = set(names)
```

- An empty set is created by passing no argument to `set()`
- Length (size) can be obtained using function `len()`

Set membership

- To check whether an elements is a member (belongs to) a set, operator **`in`** needs to be used (**`not in`** is also defined)

```
if "Luigi" in cast :
    print("Luigi is a character in Monty
           Python’s Flying Circus.")
else:
    print("Luigi is not a character in the show.")
```

Accessing set elements

- It is possible to iterate on the elements of a set using **`for`**

```
print("The cast of characters includes:")
for character in cast :
    print(character)
```

- Since sets are not sorted, accessing their elements by position, like with lists, raises an error

```
for i in range(len(cast)):
    print(cast[i])
>>> TypeError: 'set' object is not subscriptable
```

Unsorted elements

- When iterating on a set, the order in which elements are traversed depends on the internal order the container uses to store them (which is unpredictable, different than the insertion one, and may also change with different versions of Python)

Sorting the elements of a set

- To obtain a sorted representation of the elements in a set it is possible, e.g., to use function **`sorted()`** on it, which returns a list (not a set) with elements sorted in a given way (numbers in ascending/descending order, strings in alphabetical order, ...)

```
for actor in sorted(cast):
    print(actor) # actors sorted alphabetically
```

Adding elements to a set

- Sets are mutable, hence it is possible to add elements to them using method **`add()`**
- Trying to add an element that is already in the set does not have any effect (element is not added, and size of the set remains unchanged)

```
cast = set(["Luigi", "Gumbys", "Spiny"])
cast.add("Arthur")
cast.add("Spiny")
```

Removing elements from a set

- Calling method **`discard()`** removes the element passed as argument from the set (if existing)

```
cast.discard("Arthur")
```

- While **`discard()`** has no effect if the element to remove does not exists, method **`remove()`** raises an exception in that case

```
cast.remove("Arthur") # may raise an exception
```

- To remove all elements, method **`clear()`** shall be used

Subsets

- A set is a subset of another set if an only if each element of the first set is also a member of the second set
    - Method **`issubset()`** can be used to check whether the set on which it is called is a subset of the set passed as argument

```
canadian = { "Red", "White" }
british = { "Red", "Blue", "White" }
# True
if canadian.issubset(british) :
   print("All Can flag colors are in Brit flag.")
```

Equality or inequality of two sets

- Equality and inequality of two sets can be checked by simply using operators **"=="** e **"!="**
- Two sets are equal if and only if they have exactly the same members

```
french = { "Red", "White", "Blue" }
if british == french :
   print("British and French flags use
            the same colors.")
```

Union of two sets

- The union of two sets, obtained with method **`union()`**, contains all the elements present in both the sets, without duplicates

```
british = { "Red", "Blue", "White" }
italian = { "Red", "White", "Green" }
# in_either: {"Blue", "Green", "White", "Red"}
in_either = british.union(italian)
```

Intersection of two sets

- The intersection of two sets, obtained with method **`intersection()`**, contains the elements present in both the sets

```
# in_both: {"White", "Red"}
in_both = british.intersection(italian)
```

Difference of two sets

- The difference of two sets, obtained with method **`difference()`**, is a set that contains the elements of the first set not included in the second set

```
print("Colors that are in the Italian flag
        but not the British:")
print(italian.difference(british)) # {'Green'}
```

Dictionaries

- A dictionary is a container that records associations between keys and values
  - It is sometimes referred to as associative array or map
- Each key in the dictionary has value associated with it
- Keys are unique, but a value can be associated with more than one key

Defining a dictionary

- Dictionaries are created like sets using `{ }`, but passing **`key : values`** associations rather than single elements

```
contacts = { "Fred": 7235591,
             "Mary": 3841212,
             "Bob": 3841212
           }
```

- Dictionary keys need to be numbers or strings
- However, there is no limitation to the type of dictionary values

Duplicating a dictionary

- To create a copy (duplicate) of a dictionary it is possible to use function **`dict()`**

```
copyOfContacts = dict(contacts)
```

- It is also possible to create an empty dictionary

```
contacts = dict()
contacts = {}
```

- The use of **`=`** operator, in turn, does not create a copy, but an alias

Accessing dictionary elements

- The indexing operator `[ ]` can be used to return the value associated with the dictionary key passed as argument (key can be a constant, a variable or an expression)

```
print("Fred's number is", contacts["Fred"])
>>> 7235591
```

- It is not possible, instead, to access dictionary elements by index, like with a list

Checking the existence of a key

- To check whether a key is present in a dictionary, operator **`in`** (and similarly, **`not it`**) shall be used
- It is advisable to check key existence in order to avoid **`KeyError`** exceptions that are raised by methods operating on dictionaries when key does not exist

```
if "John" in contacts :
  print("John's number is", contacts["John"])
else :
  print("John is not in my contact list.")
```

Default values

- As a shortcut, to access dictionary values it is possible to use method **`get()`**, which allows to specify also a default value
- In case the key passed as first argument is not present, the method returns the default value which has been passed as second argument
  - **`number = contacts.get("Tony", "missing")`**

Adding or modifying dictionary elements

- A dictionary is a mutable container and it is possible to add an element to it using the indexing operator **`[ ]`** (this cannot be done on lists)

```
contacts["John"] = 4578102
```

- To modify the value associated with a given key it is sufficient to set the new value on the existing key using **`[ ]`**

```
contacts["John"] = 2228102
```

Removing elements from a dictionary

- To remove an element from a dictionary it is necessary to use method **`pop()`** , which receives as argument the key of the element to remove
  - The method deletes both the key and the associated value
- The element is returned, hence it is possible to store it in a variable, if needed
- If they key does not exists, a **`KeyError`** exception is raised (can be avoided by checking key existence, first)

```
if "Fred" in contacts :
    freds_number = contacts.pop("Fred")
```

Iterating on dictionary keys

- It is possible to iterate on all the keys of a dictionary with a **`for`** … **`in`** cycle

```
print("My Contacts:")
for key in contacts :
    print(key)
```

Iterating on dictionary keys

- Interestingly, starting from Python 3.6, keys are recorded in the order of insertion (hence, they are traversed in that order)
- To iterate on keys sorted in numerical or alphabetical order, it is necessary to use function **`sorted()`** in the cycle, building a temporary list of sorted key values

```
print("My Contacts:")
for key in sorted(contacts) :
    print(key, contacts[key])
```

Iterating on keys and values

- Python offers another method for iterating on dictionary elements (keys and values), named **`items()`**
  - It is more efficient than iterating on keys and possibly accessing the dictionary multiple times to get the associated values
- The method returns a sequence of tuples **`(key, value)`**

```
for item in contacts.items() :
    print(item[0], item[1])
```

- For readability, names can be assigned to tuple elements

```
for (key, val) in contacts.items() :
    print(key, val)
```

Advanced sorting

- As said, lists can be sorted using method **`sort()`** or function **`sorted()`**
- The sorting algorithms internally use multiple times the operator <, which performs a pairwise comparison
- Hence, the sorting behavior depends on the behavior of the operator < itself, which in turn depends on the type of data contained in the list
  - For lists of primitive data, numerical or alphabetical order (ascending or descending, with **`reverse`**) is used by default
  - For lists of lists or lists of tuples, the criteria for comparing lists is used by default (ascending or descending, with **`reverse`**)

Advanced sorting

- When more complex data structures are to be sorted (e.g., lists of dictionaries) or when the natural sorting criterion needs to be modified (e.g., for lists of lists), a sorting key shall be passed using parameter **`key`**

```
studenti.sort(key=...)
```

- The parameter is typically a function that can be calculated for each record (dictionary) and whose value depends only on recorded data
  - Sorting will thus be for ascending values of the sorting key

Sorting a list of dictionaries

- When working with a list of dictionaries, in the simplest cases the sorting key corresponds to one of the fields of the dictionary
- In this case, there is a function that can be used, **`operator.itemgetter()`**, which implements a sorting key to select the field of interest by passing the name of the field as argument

```
key=operator.itemgetter('name of the field')
```

Sorting a list of dictionaries

```
from operator import itemgetter
# students sorted as read from file
print(students[0], students[1], students[2])
# students sorted by studentId
students.sort(key=itemgetter('studentId'))
print(students[0], students[1], students[2])
# students sorted by name
students.sort(key=itemgetter(name'))
print(students[0], students[1], students[2])
```

Sorting a list of lists

- When working with a list of lists, i.e., a table, in the simplest cases the sorting key corresponds to one of the table columns
- As with lists of dictionaries, the function **`operator.itemgetter()`** can be used passing, as parameter, the index of the column to be used

```
key=operator.itemgetter(num_of_the_column)
```

- Similar considerations apply for lists of tuples

Using multiple sorting criteria

- Sometimes it is necessary to sort based on multiple criteria
- To this purpose, it is possible to leverage the fact that **`operator.itemgetter()`** can receive multiple arguments, corresponding to fields (for dictionaries) or indexes (for lists) to be used for sorting
- An alternative approach is to sort the data structure multiple times, each time with a different sorting key, starting from less important keys and finishing with the most important key

Predefined sorting functions

- In some cases, it is possible to use some predefined functions (basically, known functions like **`len`**, **`sum`**, etc., but without the parentheses, as they are going to be invocked by the sorting algorithm)
- For instance, to sort a list of sets based on the number of elements

```
my_sets.sort(key=len)
```

- To sort a list of lists (table) based on the sum of each row value

```
my_table(key=sum)
```

Custom sorting functions

- In the most general case, the programmer can write a custom function to calculate the sorting key, which
  - Receives an argument (an element)
  - Returns a value (the key to be used by the sorting algorithm)

```
coordinates=[[2, 3], [3, 4], [-1, 0]]
def sum_xy(p): # sort based on sum of x and y
  return p[0] + p[1]

coordinates.sort(key=sum_xy)
```