ALMA MATER STUDIORUM
UNIVERSITY OF BOLOGNA

Department of Computer Science and Engineering

Second Cycle Degree/Two Year
Master in Artificial Intelligence

Course

INTRODUCTION TO ALGORITHMS AND PROGRAMMING

Held by

Prof. Simone Martini

Written by

Giorgio De Simone

Academic Year 2021-2022

© 2023 Giorgio De Simone

Contents

Preface **1**

1 Introduction to Python **3**
1.1 The Python machine 3
1.2 Python data types 4
1.3 Assignment and Python internal state 6

2 Rudiments of programming **9**
2.1 External state 9
2.2 Logic and conditions 10
2.3 Functions 13
2.4 Bounded iterations and sequences 17

3 Values and identities **24**
3.1 Objects 24
3.2 Indeterminate iterations 27
3.3 Mutable objects 29
3.4 Recursion 56

4 Object oriented programming **59**
4.1 Classes and objects 59
4.2 Subclasses 70

5 Errors handling **80**
5.1 Exceptions 80

6 Iterators and computational complexity **94**
6.1 Iterators in-depth 94
6.2 Computational complexity of operations 99

7 Pills of NumPy **110**
7.1 Introduction to NumPy 110
7.2 Broadcasting, some final recipes 125

Appendices **136**

A Sequences **136**

B *while* and *lists* **154**

C Dictionaries and comprehesion **159**

D OOP: classes, attributes, methods, magic methods **169**

E OOP: inheritance; exceptions **177**

F NumPy **185**

Bibliography **195**

Preface

The idea of writing these notes born with the intention of collecting the entire material of the *Second Cycle Degree/Two Year Master in Artificial Intelligence* offered by *University of Bologna*, dividing it in different volumes depending on the different courses held with the purpose of creating a true series of books.

Specifically here I wrote the notes related to the *Introduction to algorithms and programming* course – held by Professor Simone Martini – which concerns rudiments of programming in Python.

Prerequisites: some programming experience. The book shall only introduce one programming language, and cannot treat programming techniques, which are largely independent from the specific language.

The book itself is essentially divided in a main section – containing the true dissertation of the lectures – and appendices – collecting a series of slides which regards the arguments of the different laboratory sessions.

Below it is shown a very preliminary schedule of the book content, schedule that is suggested to follow (but that is not mandatory for the purpose of understanding).

1. Chapter 1:
   (a) Computation as a combinatorial process. The Python machine.
   (b) Data types: integers, rationals, strings.
   (c) Assignment. The internal state.
2. Chapter 2:
   (a) The external state. `input()` and `print()`.
   (b) Booleans; conditional instructions.
   (c) Introduction to functions. Functions. Locality of names for functions.
   (d) Slices on strings. Strings as sequences. Bounded iteration (`for`). `range()` (as a sequence). Selection of elements in a sequence. `in` operator. `tuple`. Slices.
3. Appendix A:
   (a) Sequences.
4. Chapter 3:
   (a) Objects as values envelopped in an identity.

   (b) Unbounded iteration (**`while`**).
   (c) Mutable objects. **`list`**: aliasing and sharing. Shallow copies. Methods on **`list`**. **`list`** modification in the body of functions. **`list`** comprehension. Linear search in a sequence. Dictionaries and methods on dictionaries. Dictionary comprehension. Generators.
   (d) Recursion.

5. Appendix B:
   (a) **`while`** and **`list`**.
6. Appendix C:
   (a) Dictionaries and comprehesion.
7. Chapter 4:
   (a) Introduction to classes and objects. Attributes and methods. Special methods: `__init__`, `__str__`, `__add__`, `___mul__`. Examples. Examples of use of classes and objects. Equality between instances and the special method `__eq__`. Instance attributes and class attributes.
   (b) Subclasses and inheritance. Multiple inheritance. Using class hierarchies.
8. Appendix D:
   (a) OOP: classes, attributes, methods, magic methods.
9. Chapter 5:
   (a) Exceptions.
10. Appendix E:
   (a) OOP: inheritance; exceptions.
11. Chapter 6:
   (a) Iterators (and generators, again).
   (b) Representation of types in the Python abstract machine: numbers and **`list`**; cost of operations on **`list`**.
12. Chapter 7:
   (a) NumPy. Generalities: arrays, vectorisation and broadcasting. Creating arrays, reshaping. Axes. Slicing e fancy slicing. Views and copies. Stacking and concatenating. NumPy: `dtypes`. Representation of arrays: C-style and Fortran-style. Strides.
   (b) Broadcasting in-depth. Methods.
13. Appendix F:
   (a) NumPy.

At the end of the course, the reader is able to analyze a problem, formulate an algorithm for solving it, and code such an algorithm into an executable program. The reader understands some basic algorithms and a general-purpose programming language.

Chapter 1

Introduction to Python

1.1 The Python machine

What does *computation* mean in computer science?

Definition 1.1.1 (**Computation**).
Computation is a combinatorial *manipulation* of symbols from a finite alphabet. *Manipulation* is done through simple, combinatorial, effective elementary operations.

A paradigmatic (numerical) computation is the addition algorithm learned in second grade. But there are also non numerical computation.

Example 1.1.2 (**Non numerical computation**).
An example of non numerical computation is the following:

```
BOLOGNA
ANGOLOB
```

Computations are *performed* by *machines*.

Definition 1.1.3 (**Machine**).
We call *machine* the agent performing the computation.

So, the machine for a computation must be able to perform *elementary operations* for whiches the computation is built on.
To specify a computation we must specify the elementary operations and the *sequence* of those operations that should be applied.
So we have a *description* of the computation. This description is done through a *certain language*. And the machine should be able to '*understand*' (better: *execute*) that language.

Definition 1.1.4 (**Programming language**).
A (programming) *language* is something describing computations as sequences of elementary operations to be applied on some data.

We have *machines* able to execute those descriptions.

Definition 1.1.5 (**Abstract machine**).
Let suppose that we have a language for computations L. We call *abstract machine* for L any agent able to execute the computations described in L (the *programs* written in L).

There are machines that are able to compute *any* computable function. These are the machines whose languages are *general purpose programming languages*. One of these machines is the *Python machine*, which we will describe in these notes.

Definition 1.1.6 (**The Python machine**).
The Python machine is an agent able to execute any description of a computation written in the *Python programming language*.

1.2 Python data types

Elementary operations act on data. Data are organized in types. A type has a set of values (and not only these).
One of the most symple type is the *integer type*.

Example 1.2.1 (**Integer type**).
The name of integer type inside the language is **`int`**.

- It has *values*: the integers, as in Math;
- *presented* as in Math: sequences of digits, possibly with a sign:

```
+123
−678
```

- It has *operations i.e* the standard ones, with their standard presentation:

```
+ − * ** // %
A // B
A = (A // B) * B + (A % B)
+ # (unary)
− # (unary)
```

- *Precedentes* are the same as in standard Math; they can be changed by using parenthesis. In general, *computation is always performed left to right*, when possible. In order of decreasing precedence:

```
parenthesis
**
+ − # (unary)
* // %
+ − # (bynary)
```

Onother most symple type is the *string type*.

Example 1.2.2 (**String type**).
The name of string type inside the language is **`str`**.

- *Values*: finite sequences of digits, letters, other simbols taken from a certain alphabet as fixed in the official documentation;
- *presentation*: the sequence enclosed in quotes: ', ' ', ' ' ';
- *operations*: concatenation, length, repetition, selection.

Remark 1.2.3 (Operator overloading).
The operator + is used both for *addition* and *concatenation*.
How does Python disambiguates between the twos? The choiche depends on the type of the operands (mostly of the time is the type of the first operand).
This is called operator *overloading*.

Now we are ready to define what is a *type*.

Definition 1.2.4 (Type).
A *type* is a pair:

- a *set* of effectively presented *values*;
- a *set* of effective, elementary *operations* on those values.

We have seen the types `int` and `str`.

Remark 1.2.5 (Defining a type).
For any type we must give:

- its *values*;
- how those values are *presented*;
- which *operations*;
- how those operations are *presented*;
- the *name* of the type inside the language.

Remark 1.2.6 (Converting values type).
The names of the types (*e.g.* `int`, `str`) can be used to *convert* values into that type (when this makes sense):

```
str(123)
int('234')
int('bologna') # will result in an error
```

So, if operations and values interact, what is the result?

Definition 1.2.7 (Expressions).
Expressions are sequences of operations on values, using operators and parenthesis.

Example 1.2.8 (Expression).
Just a very simple expression:

```
len(str(123)) + 2 ** (len('BO') + 1)
```

A third symple type is the *float type* (rational numbers).

Example 1.2.9 (**Rational type**).
The name of rational type inside the language is `float`.

- *Values*: a subset of the rational numbers (the specific subset is defined by the specific Python machine).
- *presentation*: decimal rational representation: `3.1415`, `-3.1415`, `3.4567e3` means `3.4567 * 10 ** 3`;
- *operations*: standard math operations:

```
+ - # (unary)
+ - * / **
```

Remark 1.2.10 (**Mixing 'int' and 'float'**).
We may mix `int` and `float`. Python machine will apply the necessary transformations of types so that the expression makes sense (*e.g.* `7 / 3` is equivalent to `float(7) / float(3)`).

1.3 Assignment and Python internal state

Definition 1.3.1 (**Assignment**).
An *assignment* in Python has the following sintax:

```
name = expression
```

Where:

- `name` is a (finite) sequence of letters, digits, and `_`, which does not start with a digit;
- `expression` see Definition 1.2.7.

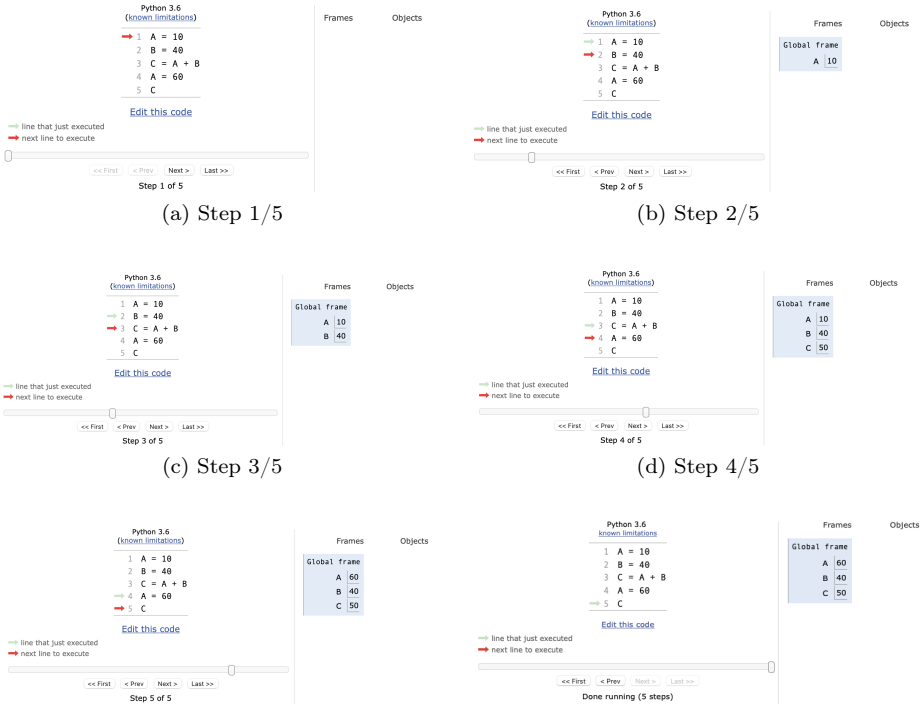
The assignment is that elementary statement of the language which associates a name with a value.

Remark 1.3.2 (**Assignment semantics**).
The assignment:

```
name = expression
```

1. Evaluate `expression`, determining a value `v`;
2. check if `name` is already present;
3. if not present, create it;
4. bind `v` to `name`.

The Python machine mantains an *internal state*.



(a) Step 1/5 (b) Step 2/5

(c) Step 3/5 (d) Step 4/5

(e) Step 5/5 (f) Done running (5 steps)

Figure 1.1: evolution of the internal state.

Definition 1.3.3 (Internal state).
The Python machine mantains a *list* of *associations (bindings)* between names and values.
This list is called the *internal state* of the machine.

Remark 1.3.4 (Internal state evolution).
We assume for the moment that when we start a Python machine, its internal state is empty.
Then any assignment on a *new* name will add a line to the internal state.
An assignment on a *existing* name changes the binding for that name *only*.

Example 1.3.5 (Evolution of the internal state).
The internal state evolves as an ordered list: new names are added at the end of the list.
Assignments to already used names modify the binding for those names *only*.

```
>>> A = 10
>>> B = 40
>>> C = A + B
>>> A = 60
>>> C
50
```

See Figure 1.1 to have a pratical view of the evolution.[1]

Let's make clarity of some concepts seen so far.
How *types* are related to *values* and *names*?

Remark 1.3.6 (**Types are an attribute of values**).
Types are an attribute of *values*, not of *names* (the type of a value bound to a name may change freely).
We may inspect the type of a value by usign the operation (function) `type()` (*e.g.* `type("Hello")`).

What is the difference between *expressions* and *commands*?

Remark 1.3.7 (**Expressions vs commands**).
An *expression* is a phrase in the language of which we are interested in its *value*:

```
12 + 3 # return 15
len('bologna') + 1 # return 8
```

A *command* is a phrase in the language which we use for its modification of the (internal) state:

```
A = 10
```

Finally, we can answer to a main question of computer science: what is a (Python) *program*?

Definition 1.3.8 (**Python program**).
A *program* is a text. Any line of this text is a single Python phrase (command or expression).
We may ask a Python machine to *run* a program (to evaluate, to execute). The machine will execute the program, line by line, starting with line 1, then line 2 ... until the end of the text is reached.

[1] We used *pythontutor* (`https://pythontutor.com`) to represent this evolution.

Chapter 2

Rudiments of programming

2.1 External state

In the previous chapter we mentioned the *internal state* related to the Python machine; then an internal state implies both the existence of an *external state* and the possible *interaction* between the two states themselves.

Example 2.1.1 (External/Internal state interaction).
An example of interaction between internal and external state are the following:

- `print()` is a *command* which transfers the values of its arguments to the *external state*;
- `input()` is an *expression* which transfers values from the *external* to the *internal state*. `input()` always returns values of type `str`.

Example 2.1.2 (Swapping two values).
For swapping the bindings (values) of two names (using only the part of the language that we have seen so far) we must use a *third* name:

```
A = 10
B = 200
# (swapping)
tmp = B
B = A
A = tmp
```

We strongly suggest to visualize the effect of this code on the *internal state* (through *pythontutor*).

Remark 2.1.3 (Strings/Integers/Rationals are immutable).
Strings are immutable. One may be tempted to write:

```
S = 'bologna'
S[0] = 'c'
```

This is an error: we cannot modify a *value* of type `str` (like we cannot modify a value of type `int` or `float`).
Of course we may modify the binding of a name:

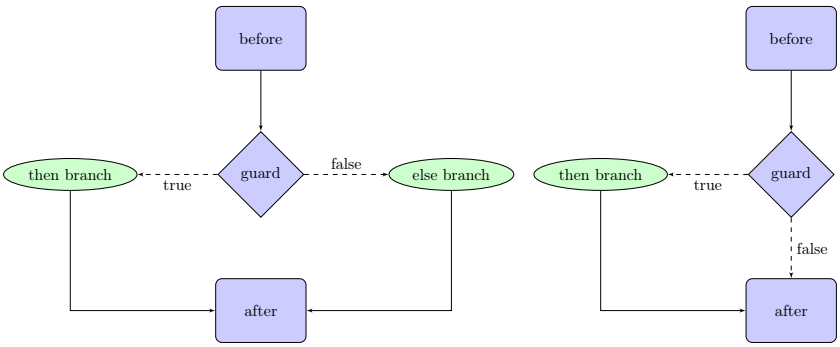


Figure 2.1: conditional command first form: 'if - else'.

```
S = 'cologna'
```

but the two values 'bologna' and 'cologna' exist independently and distinct.

2.2 Logic and conditions

We can introduce some logic and conditions into the code.

Remark 2.2.1 (**Conditional command**).
A *conditional command* is a compound command: it is composed (also) from other commands.
Its use: altering the sequential order of execution of a program, according to some *conditions*.

Definition 2.2.2 (**Reserved name**).
A *reserved name* (or *reserved word*) has the structure of a standard name but *cannot be used as a name* because of its special role inside the language.

Definition 2.2.3 (**Conditional command first form: 'if - else'**).

```
if <condition>:    # guard
    <t-block>      # then branch
else:                                # optional
    <e-block>      # else branch     # optional
```

Where `<condition>` is (syntactically) any expression of type **`bool`** (see next Definition).

- *Syntax*: **`if`** and **`else`** are *reserved names*;
- *semantics* (see Figure 2.1): we evaluate the *guard*. If the guard is true, we execute the *then branch*; if the *guard* is false, we execute the *else branch*. After the *then/else branch* (only one of those!) the execution proceeds from what follows the conditional command. Note that the **`else`** (and its branch) is optional, meaning that if is present only the **`if`** condition and it is false, then the execution flow proceeds linearly to the following command.

So, at this point, we must introduce a new type:

Definition 2.2.4 (Boolean type).
Another Python type is the **bool** type.

- *Values*: only two;
- *presented* as: `True`, `False`;
- *operations*: **and**, **or**, **not**. Where:

```
E1 and E2
# this is True if and only if both E1, E2 are True

E1 or E2
# this is False if and only if both E1, E2 are False

not E1
# this is True if E1 is False
```

Mainly we have two kinds of *operators*:

- the *logical* ones (just introduced above) which already work with the boolean values `True`/`False`;
- the *comparison* ones which *produce* boolean values comparing values of other types.

Definition 2.2.5 (Logical operators).
Logical operators:

- **not** (negation): it transforms `True` into `False` and `False` into `True`;
- **and** (conjunction: et): `<exp1>` **and** `<exp2>` has value `True` if and only if both `<exp1>`, `<exp2>` have value `True`;
- **or** (disjunction: vel): `<exp1>` **or** `<exp2>` has value `False` if and only if both `<exp1>`, `<exp2>` have value `False`.

Definition 2.2.6 (Comparison operators).
The *comparison operators* produce boolean values comparing values of other types. They are:

```
# (reading left to right)
<   # less than
<=  # less than or equal
>   # greater than
>=  # greater than or equal
==  # equal
!=  # not equal
```

Remark 2.2.7 (Evaluation of logical operators).
Python expressions are *completely* evaluated from left to right:

```
7 * 0 * (int(input()))
```

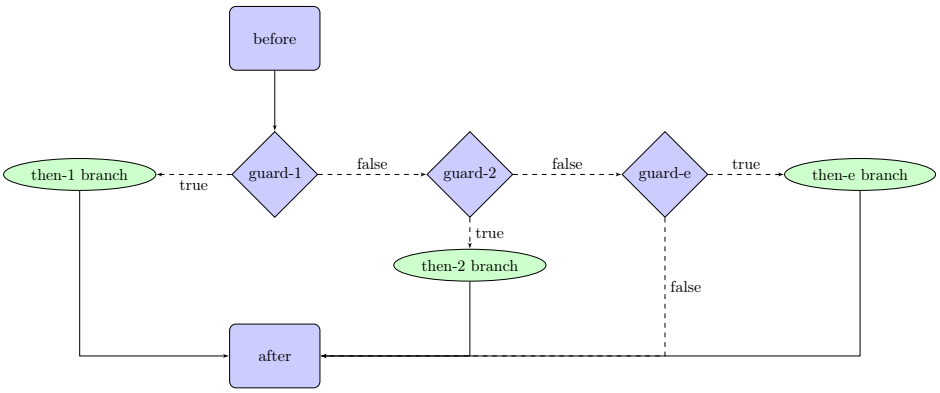


Figure 2.2: general form of the conditional: 'if - elif - else'.

The `input()` (at the extreme right of the expression) is always performed, even if the result `0` is already known.
Boolean expressions are an exception to this. Boolean expressions have a *lazy* evaluation: as soon we find a value that allows to know the final result, the evaluation terminates:

```
(0 == 0) and (0 == 1) and (0 == int(input()))
```

the evaluation terminates after the value of `(0 == 1)` which is `False`, because the `and` with a `False` is always `False`. Hence no call to `input()` happens.

We can have a more general form of `if` - `else` conditional command, utilizing *nested* `if`: we may use `elif`.

Definition 2.2.8 (**General form of the conditional: 'if - elif - else'**).

```
if <expr_1 bool>:
    <block_1>
elif <expr_2 bool>: # optional
    <block_2>
...

elif <expr_n bool>: # optional
    <block_n>
else:               # optional
    <block_e>
```

See Figure 2.2 for a graphical logic view.

In the previous definition we mentioned `<block_1>`, `<block_2>`, ... `<block_n>`; but what do we mean with the word *block*?

Definition 2.2.9 (**Block**).
With the term *block* we mean:

- a sequence of lines;
- each with a single command;

- all of them at the same *indentation* (distance from the left margin).

It is used to *structure* execution[1] together with compound commands (as **if**).

Example 2.2.10 (**Blocks**).
How many blocks?

```
x = int(input())                        #
if x != 0:                              #
    y = 10        # -> block_1          #
    print(y)      #                     #
else:                                   # -> block_0
    z = 20        #                     #
    print(z)      # -> block_2          #
    x = 100       #                     #
print(x)                                #
```

2.3 Functions

Example 2.3.1 (**Functions: definition and use**).

```
# DEFINITION
def perimeter(b, h):
    return 2 * b + 2 * h

def area(b, h):
    return b * h

# USE/CALL
print(area(10,2))

print(perimeter(3,6))

print(area(10, perimeter(3, 6)))
```

Definition 2.3.2 (**Functions: definition and use**).
Functions are present in the language in two ways:

1. *definition* of the function:

   ```
   def <name>(<parameters>):
       <body of the function>
   ```

   When a function is defined, a binding between its name and its body is inserted into the internal state. No computation is done;

2. *use (call)* of the function:

   ```
   <name>(<arguments>)
   ```

[1] Other programming languages use parenthesis {} or 'begin - end' pairs. Python uses *indentation*.

When a function is used (called) the *values* of the arguments are bound to the parameters, and the body is executed. When during the execution of the body the **return** is evaluated, the function terminates and the value of the expression after **return** is given as value of the call.

Let's investigate more accurately what we introduced with the previous Definition.

Remark 2.3.3 (**Functions: definition**).

```
def <name>(<formal parameters>):        # header
    <block>                             # body
```

Where:

- **def** is a reserved name; we alredy saw the reserved names **if** and **else**. We will see many others;
- `<formal parameters>` are comma separated names;
- `<block>` is a sequence of commands all at the same indentation.

A **def** is a compound command (like **if**).

Remark 2.3.4 (**Functions: use**).
To use (call) a function the following syntax is adopted:

```
<name>(<arguments>)
```

Which is an expression. Where:

- `<arguments>`, also called `<actual parameters>`, is a comma separated list of expressions (in the same number of the `<formal parameters>` in the header of the function).

Remark 2.3.5 (**Function: 'return'**).

```
return <expression>
```

Seen in the Example 2.3.1 and mentioned in the Definition 2.3.2, is a command that may appear only inside the body of a function. When is executed, it forces the termination of the execution of the body; the value of `<expression>` is *returned* as the value of the function call. **return** is a reserved name. Note that `<expression>` can be blank, so it is optional; in that case the **return** is used just to quit from the body.

Remark 2.3.6 (**Function: semantics**).

```
def f(x):
    body
```

A definition (**def**) of a function introduces into the internal state a binding between the name of the function and its body.

```
f(exp)
```

When a function is called:

1. `exp` is evaluated to a value `v`;
2. in the internal state (in the local frame for `f`), `x` (the formal parameter of `f`) is bound to `v` (the value of the actual parameter);
3. `body` is executed in this state;
4. the evaluation of any **`return`** `<exp_2>` causes the termination of the function. The value of `<exp_2>` is *returned* as value of the call.

We have seen so far two different kinds of parameters: ones that are written during the definition phase and the others that can be passed during the call. Let's define more precisely the concepts.

Definition 2.3.7 (**Formal/Actual parameters**).

- `Formal parameters` (or `parameters`, for short) are `names`; comma separated if more than one. If there are no parameters, then the **`def`** has the form:

```
def f():
    body
```

- `Actual parameters` (or `arguments`, for short) are `expressions`. They must be in the *same number* of the `formal parameters`. At the moment of the call they are bound to the `formal parameters`, respecting their order.

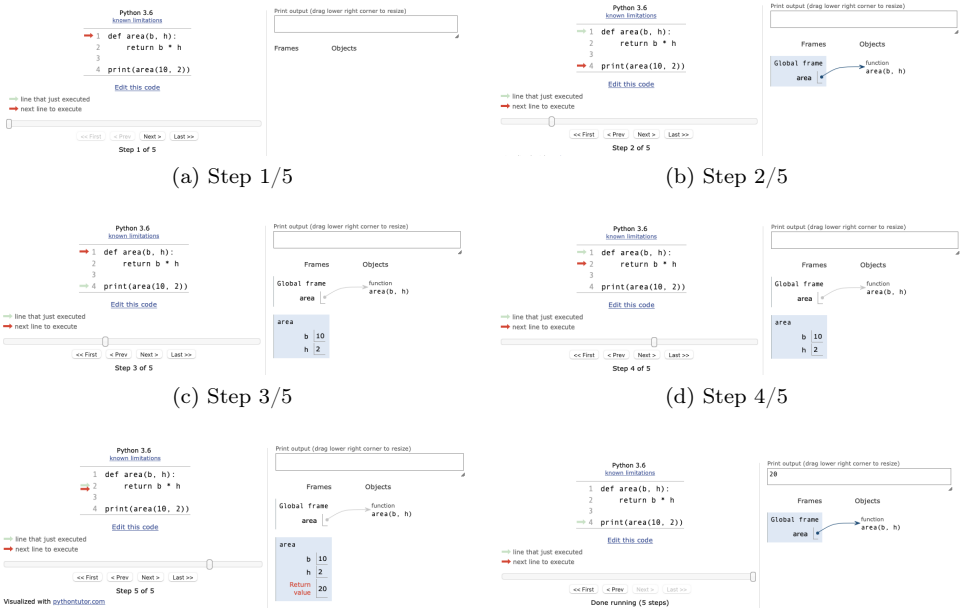
Remark 2.3.8 (**Functions and the internal state**).
As we can see in the Figure 2.3 the behaviour of the internal state related to a function is essentially the following:

- any call to a function creates a new frame in the internal state;
- when a function **`return`**, its frame is erased from the state;
- frames are added and removed from the state like dishes from a stack of dishes. Hence the name for the internal state: the `frame stack`, or the `call stack`;
- any name occurring to the left of an assignment in the body of a function is *local* to that function: the binding is created inside the frame for that function and the binding is deleted together with frame, at **`return`** time.

Example 2.3.9 (**Functions and the internal state**).
Check on *pythontutor*:

```
# DEFINITION
def perimeter(base, h):
    P = (base + h)
    return P
    return 100      # never executed

def z(h):
    return 0
```



(a) Step 1/5

(b) Step 2/5

(c) Step 3/5

(d) Step 4/5

(e) Step 5/5

(f) Done running (5 steps)

Figure 2.3: function evolution of the state.

```
def f(x):
    N = z(1) + 1 + x
    return N + 1

# USE/CALL
>>> Per = 0
>>> Per = perimeter(10, 2)
>>> W = f(9)
>>> print(Per, W)
12 11
```

In order to conclude the current paragraph, we make the following brief recap.

Remark 2.3.10 (**Functions: recap**).
So far, regarding functions, we have seen:

- definition vs use;
- local frame for each function call;
- **return** and its meaning;
- any name appearing in the body of a function to the left of an assignment is local to that function.

2.4 Bounded iterations and sequences

How can we behave if we want to iterate some actions?

Example 2.4.1 (**Iteration: primer**).
Given the string 'bologna' we want to print its elements one by one, on a new line:

```
s = input()
print(s[0])
print(s[1])
print(s[2])
print(s[3])
print(s[4])
print(s[5])
print(s[6])
```

But the way seen above is not the best for bounded iterations; for a word of 24 characters we should write the same command 24 times. We should use the **for** construct.

Example 2.4.2 (**Boundend iteration: 'for'**).

```
s = 'bologna'
for c in s:
    print(c)
```

This construct express a repetition: in this case the command **print(c)** is executed on all elements (all characters) of (the value of) s.

Definition 2.4.3 (**'for' construct**).

```
for <name> in <sequence>:      # header of the iteration
    <block>
```

Where **for** is a compound command (as **if**) and regarding <sequence> for the moment we know only the string type.

- Semantics:
  1. <sequence> is *computed and frozen*;
  2. if <name> does not exist, it is created;
  3. <name> is bound to the *first* element of the <sequence>;
  4. evaluate <block>;
  5. if there is a next element in <sequence>, bind <name> to this element and repeat from step 3.;
  6. if there is not a next element in <sequence>, the evaluation of the command terminates (and hence the execution passes to the next line of the program).

We have just seen the reserved word **in** used in the header of the iteration construct **for**, but it can be also a *boolean operator*.

Definition 2.4.4 (**A boolean operator: 'in'**).

```
<element> in <sequence>
```

It is a boolean expression with value `True` *if and only if* `<element>` is an element of `<sequence>`.

```
'a' in 'aeiou'        # has value True
'p' in 'simone'       # has value False
```

Of course both `<element>` and `<sequence>` may be arbitrary expressions.
We highlight that *only if* `<sequence>` is a string we have as result of the expression: `True` if `<element>` is a substring of `<sequence>`.

Remark 2.4.5 (**The two uses of the 'in' operator**).
Do not confuse the two uses of the reserved name `in`:

```
for <name> in <sequence>:
```

is the header of an iteration.
While:

```
<element> in <sequence>
```

is a boolean expression.
We cannot mix the twos. Examples:

```
for s in 'simone' and s in 'martini':      # NO!
```

We may combine logical expressions:

```
if 's' in 'simone' and 's' in 'martini'
```

The types seen so far `int`, `float`, `bool`, `str`, `NoneType`, are *simple type*. Now we are going to define a new kind.

Definition 2.4.6 (**A new type: 'tuple'**).
The `tuple` type is a *structured type*: its values are groups of values (each of them of arbitrary type). A `tuple` is an *immutable sequence* of objects.

- *Values*: finite sequence of values (each of them of arbitrary type);
- *presentation*: `(object_1, ..., object_k)` or also without parentheses because is *the comma* the constructor for tuples;
- the `tuple` with a single element is `(element,)`;
- *operations*: concatenation (`+`), repetition (`*`), length (`len()`), selection (`[...]`). They are the common operations on most other types of sequences (like for example strings);
- *indexes*: like for `str` *i.e.*
  - they start from `0`;
  - negative indexes count from the end.

Example 2.4.7 (**A new type: 'tuple'**).
Examples of **tuple**:

```
(10.3, 100, 'simone')
(3,)
()
(1, 2, (100, 200), 3)
(10, 20,)
10, 20
```

Non examples of **tuple**:

```
(3)
(10,, 20)
(10, , 20)
[10, 20]
(,0)
```

Remark 2.4.8 (**Iteration on 'tuple'**).
We may perform a **for** on a **tuple**.

```
for e in (1, 2, 'bologna', (3, 4)):
    print(e)
```

Example 2.4.9 (**Select and name elements**).
Try them in *pythontutor*:

```
T = (1, 2, (100, 200), 3)
X = (10, 20)
W = (9, X, 11)
Z = T[2]
```

Remark 2.4.10 (**Aliasing on 'tuple' is harmless**).
Aliasing is possible: different names (*the aliases*) refer to the same object.
But we highlight that aliasing on **tuple** is harmless, because **tuple** is immutable.

We can use **tuple()** as type converter.

Remark 2.4.11 (**Type converter: 'tuple()'**).
As other names of types, **tuple()** may be used to convert values into tuples.

```
>>> tuple()
()
>>> tuple('bolo')
('b', 'o', 'l', 'o')
>>> tuple((1, 2, 3))
(1, 2, 3)
>>> tuple(100)
TypeError: 'int' object is not iterable
>>> tuple(range(2,6))
(2,3,4,5)
```

Definition 2.4.12 (**Generalised assignment**).
We already saw the standard assignment:

```
<name> = <expression>
```

Generalised:

```
<name_1>, ..., <name_k> = <sequence with k elements>
```

For its meaning, recall how an assignment is evaluated:

- evaluate the RHS[2] and obtain its value;
- bind this value to the name(s) in the LHS.

Example 2.4.13 (**Generalised assignment**).

```
A, B = (1, 2)
C, D, E = (A + 1, 5, B)
C, E = (C + 1, E + 1)
F, G = "pr"

A, B = B, A # The Pythonic way of swapping two bindings
```

Definition 2.4.14 (**Slice**).
As a first approximation we can use *slice* as selection of a portion of a sequence:

```
<sequence>[<first_index> : <last_index>]
```

- *Semantics*: subsequence of `<sequence>` starting at `<first_index>` (included) up to `<last_index>` (not included);
- *warning*: `<first_index>` < `<last_index>`; otherwise the sequence is empty.

Example 2.4.15 (**Slice**).

```
>>> s = 'bologna BO'
# Who are they?
>>> s[0 : 4]
'bolo'
>>> s[3 : -1]
'ogna B'
>>> s[3 : 2]
''
>>> s[3 : 4]
'o'
>>> s[3 : 3]
''
```

Remark 2.4.16 (**Implicit slice**).
We can also leave blank indexes in a slice with the following meaning:

[2]RHS stands for 'right-hand-side' while LHS for 'left-hand-side'.

```
<sequence>[: <end>] # start at 0

<sequence>[<start> :] # ends at len(<sequence>) + 1,
                      # last element included

<sequence>[:] # all
```

Who are they?

```
>>> s = 'bologna BO'
# Who are they?
>>> s[: 4]
'bolo'
>>> s[3 :]
'ogna BO'
>>> s[:]
'bologna BO'
>>> s[: -1]
'bologna B'
```

Remark 2.4.17 (**Slice is not a generalised selection**).
Warning: a slice is a *window* over the sequence:

```
>>> s = 'bologna BO'
'bologna BO'
>>> s[6 : 12]
'a BO'
>>> s[12]
IndexError: string index out of range
```

Therefore `s[i : i + 1]` is not equivalent to `s[i]`! (If `i` is not a valid index, the first evaluates to the empty sequence, the latter is an error).

Definition 2.4.18 (**Slice: step**).

```
<sequence>[<first_index> : <last_index> : <step>]
```

- *Semantics*: subsequence starting at `<first_index>` and terminating at `<last_index> − 1`, each element distant `<step>` elements from the preceding one.

```
<sequence>[<first_index> : <last_index>]
# is a shorthand for
<sequence>[<first_index> : <last_index> : 1]
```

Example 2.4.19 (**Slice: step**).

```
>>> s = 'bologna BO'
# Who are they?
s[1 : 9 : 2]
'oon '
>>> s[: 100 : 4]
'bgB'
```

```
>>> s[3 : : 3]
'oaO'
>>> s[3 : -1 : 3]
'oa'
```

Remark 2.4.20 (**Slice: negative step**).
When the <step> is negative, the subsequence is extracted from the end, and <start> > <end>.

```
>>> s = 'bologna BO'
>>> s[5 : 2 : -1]
'ngo'
>>> s[ : 3 : -1] # end of the sequence (last included!)
'OB ang'
>>> s[6 : : -1] # start (0 included!)
'angolob'
>>> s[ : : -1]
'OB angolob'
>>> s[8 : 2 : -3]
'Bn'
>>> s[8 : -1 : -1] # warning
''
```

Definition 2.4.21 (**Special sequences: 'range()'**).

```
range(<start>, <end>, <step>)
```

This generates a sequence of integers between <start> and <end> − 1, each <step> apart from the previous.
Let's make some considerations one **range()**:

- + and ∗ are not defined on ranges;
- **range**(<end>) is a shorthand for **range**(0, <end>, 1);
- if <end> < <start> range is empty;
- negative steps: same as for slices;
- we may do **for** on a range.

We conclude this paragraph with the following remark.

Remark 2.4.22 (**Iterations and sequences: recap**).
Basically we have seen in this paragraph:

- sequences, in particular:
  - strings;
  - tuples;
  - ranges.
- Operations on sequences:
  - concatenation;

  - repetition;
  - lenght;
  - selection;
  - slices.

- *Intended* control statement on sequences:
  - **`for`** command.

Chapter 3

Values and identities

3.1 Objects

Remark 3.1.1 (**Objects and operations**).
Operations create new objects.
On integers is obvious:

```
x = 8
x = x + 1
```

In the second assignment, we RHS evaluates to `9`, which is bound to name `x`. `9` is a new value, is not (obviously!) a modification of 8 (which does not make any sense).
The same is true for other types! And then it appears less obvious.
On tuples:

```
x = (10, 20, 30)
y = x
x = x + (40, )
```

In the last assignment, we RHS evaluates to `(10, 20, 30, 40)` which is bound to name `x`. `(10, 20, 30, 40)` is a new value, is not a modification of `(10, 20, 30)` (which does not make any sense).

We talked about *objects* but we have no yet defined the concept properly.

Definition 3.1.2 (**Object**).
An *object* is a *value envelopped in an identity*. This identity is unique during a run of the Python machine.
In Python *any value is an object* (even `int` values).
We may have distinct objects (with different identities) and same value.
The identity of an object is completely controlled by the Python machine: we may inspect the identity of an object by the function `id()`, but we cannot change it.

Definitioon 3.1.3 (**Identity vs equality**).
The operator == is equality between *values* (nothing to do with identities):

```
value_1 == value_2
```

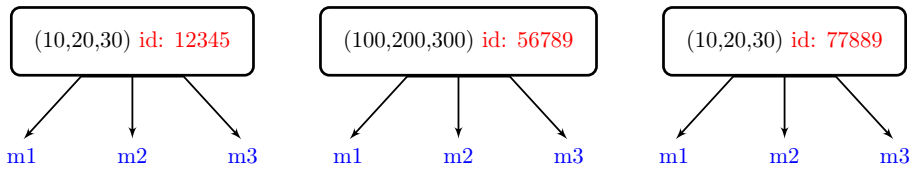


Figure 3.1: object envelops (identities and methods).

The operator **is** is the comparison operator; its response is `True` if and only if *identities* are the same:

```
obj_1 is obj_2
```

Obvious: two identical objects for **is**, are also ==; the viceversa is not true in general.

A this stage, the **is** relation is not much relevant, it will become relevant when *mutable* objects will be introduced.

Remark 3.1.4 (**Objects: identity**).
Important: we can never assume identity of objects from the fact that they have the same value.
On the contrary we must always assume that: *any operation creates new objects*.

The envelope enclosing an object contains, besides its identity, also other informations; an important information of an object is its *methods*.

Remark 3.1.5 (**Objects: methods**).
An object is an *active value*: we may stimulate the object through the *methods* offered by its envelope.
Objects of the *same type share the same methods*; see Figure 3.1.
Terminology: a method is *invoked* on an object or it is *sent* to the object or also (we like this less) it is *called* on the object.
Methods act on the object on which they are invoked; they stimulate the object to do something (*e.g.* to return a value).
The *syntax* to invoke a method is the following:

```
<object>.<method>(<optional arguments>)
```

Recall: the object receiving the method can be a constant, a name, an expression *etc.*; in summary can be any Python expression evaluating to an object.

Example 3.1.6 (**Objects: methods**).
The type **str** offers the method `upper()`:

```
s = 'bologna BO'
t = s.upper()
print(t)
```

Definition 3.1.7 (**Some methods offered by strings**).
Some methods of type **str**:

- methods without parameters:

  - `str.capitalize()`:
    * **return** a copy of the string with its first character capitalized and the rest lowercased;
  - `str.isalpha()`:
    * **return** `True` if all characters in the string are alphabetic and there is at least one character, `False` otherwise;
  - `str.isdigit()`:
    * **return** `True` if all characters in the string are digits and there is at least one character, `False` otherwise;
  - `str.upper()`:
    * **return** a copy of the string with all the cased characters converted to uppercase;
  - `str.lower()`:
    * **return** a copy of the string with all the cased characters converted to lowercase.

- Methods with parameters (observe the notation used by the official documentation):
  - `str.find(sub[, start[, end]])`:
    * **return** the lowest index in the string where substring `sub` is found within the slice `s[start : end]`. Optional arguments `start` and `end` are interpreted as in slice notation. Return -1 if `sub` is not found.[1]
  - `str.count(sub[, start[, end]])`:
    * **return** the number of non-overlapping occurences of substring `sub` in the range `[start, end]`. Optional arguments `start` and `end` are interpreted as in slice notation. If `sub` is empty, returns the number of empty strings between characters which is the length of the string plus one.

Definition 3.1.8 (**Some methods offered by sequences**).
Some methods offered by sequences, hence – for example – by **`str`** and **`tuple`** (sequences seen so far):

- `s.index(x)`:
  - **return** the index of the first occurence of `x` in `s`. It is an error if `x` is not an element of `s`;
- `s.count(x)`:
  - **return** total number of occurences of `x` in `s`.

[1] The sintax used for the method parameters is read in this way: the argument `sub` is mandatory; after it begins `[, ]` which means that the parameter `start` inside the parenthesis is optional referring to `sub`; after it begins another `[, ]` which means that the parameter `end` is optional referring to `start`, meaning that – for example – can't be present as arguments of the method only `sub` and `end` – leaving `start` empty – because `end` is optional only referring to `start`. Briefly the optional dependency is linear and not transitive.

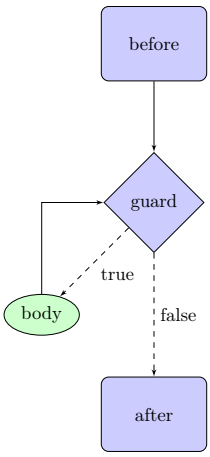


Figure 3.2: indeterminate iteration command: 'while'.

Definition 3.1.9 (**Some methods offered by 'float'**).
Some methods offered by rationals *i.e.* by `float`:

- `fl.as_integer_ratio()`:
  - `return` a pair of integers whose ratio is exactly equal to the original float and with a positive denominator;
- `fl.is_integer()`:
  - `return` `True` if the `float` instance is finite with integral value and `False` otherwise.

3.2 Indeterminate iterations

Not always is possible to solve problems utilizing bounded iterations, meanining that – for example – we can't know a-priori how many times we need to perform an action in order to reach the scope.
To give a more precise idea consider the following.

Example 3.2.1 (**Necessity of indeterminate iteration**).
Problem: we randomly extract integers from `0` to `9` and count. Given an integer `x` (between `0` and `9`) we want to count how many extractions we perform before extracting `x`.
We don't know *beforehand* how many extractions nedeed to solve this problem. Bounded iteration is not the right construct.

At this point we can introduce another construct capable to perform indeterminate iterations.

Definition 3.2.2 (**'while' command**).

```
# <before>
while <bool-expr>: # guard
      <block> # body
# <after>
```

- Semantics (see Figure 3.2):
  1. evaluate <**bool**–expr> (the <guard>);
  2. if the <guard> is `False`, terminate the command (*i.e.* proceed with the command following the <block>), at the indentation of **while**: *we exit from* **while**;
  3. if the <guard> is `True`, evaluate <block> (the <body>): *we enter inside the* **while**. Go back to step 1.
- Prerequisites:
  1. <before> must garantee an *initial value to* <guard>;
  2. <body> must modify the <guard>.

A **while** command is never alone: in the <before> part should be initializations, <body> must modify the <guard>. These are not grammatical requirements but good programming practise.

We can express bounded iterations through indeterminate iterations. We can express **for** through **while**.

Remark 3.2.3 (**Command 'for' expressed through command 'while'**).
We can express **for** through **while**:

```
for i in range(len(s)):
    <body>
```

With some approximation correspond to:

```
i = 0
while i < len(s):
    <body>
    i = i + 1
```

This last construct does not take into account the *freezing* of the sequence and the restrictions on the control name.

But it is evident that the **for** construct is more concise in comparison to the **while** construct. So, also if we can express **for** through **while**, it should be more favorable to use directly **for**, when possible.

Remark 3.2.4 (**'for' is more favorable then 'while'**).
Compare the followings:

- using **for**:

  ```
  somma = 0
  for e in s:
      somma = somma + e
  print(somma)
  ```

- Using **while**:

```
i = 0
somma = 0
while i < len(s):
    somma = somma + s[i]
    i = i + 1
print(somma)
```

So every time is possible, it should be used:

- **for** without indexes (directly on the sequence);
- **for** instead of **while**.

Example 3.2.5 (Infinite loop).
Consider the following program and ask yourself if it is a correct program:

```
while 1 == 1:
    x = 100
print(x)
```

We can say that is:

- *syntactically* correct;
- *semantically* wrong: it is an infinite loop which never interacts with the external world.

3.3 Mutable objects

Now we introduce a new data type. But before we need to explain what means being *mutable*.

Definition 3.3.1 (Mutable object).
Being *mutable* for a Python object means that:

- we may *modify the value* of an object;
- preserving its identity.

Definition 3.3.2 (A new type: 'list').
The **list** type is a *structured* type – like **tuple** – but differently from **tuple** – and contrary to all types seen so far – it consists in a *mutable sequence* of objects.

- *Values*: finite sequence of values, *mutable* (see Definition 3.3.1);
- *presentation*: `[object_1, ..., object_k]`;
- the **list** with a single element is `[element]` or also `[element, ]`; `[]` is the empty **list**;
- *operations*: concatenation (+), repetition (∗), length (**len()**), selection (`[...]`);
- *indexes*: like for **str**.

Example 3.3.3 (**Type 'list' is mutable**).
Example:

```
>>> L = [10, 20, 30]
>>> id(L)
140731267442320
>>> L[1] = 200
>>> L
[10, 200, 30]
>>> id(L)
140731267442320
```

The object bound to `L` did not change; its value changed. Warning: LHS of = is not a name (regarding `L[1] = 200`).

With mutable object we can introduce a new kind of *assignment*.

Definition 3.3.4 (**Assignment on selection**).

```
<selection> = <expression>
```

Where `<selection>` must be on *mutable* object (and cannot be a name).

- Semantics:
  - evaluate `<expression>` and obtain an object `o`;
  - evaluate `<selection>` and obtain a component `p` of a modifiable object `m`;
  - the component `p` of `m` is modified in `o`.

Observe: there is *no* modification of the *bindings between names and values* in the internal state

Remark 3.3.5 (**Different types of assignment**).
The assignment seen so far is:

```
<name> = <expression>
```

And its general form:

```
<name_1>, ...,  <name_k> = <sequence with k elements>
```

Now we introduced:

```
<selection> = <expression>
```

That can be generalized in:

```
<select_1>, ...,  <select_k> = <sequence with k expr>
```

Where if `<select_k>` select more than one element, then the corrispondent `<expression_k>` has to be an *iterable*.

Example 3.3.6 (**Different types of assignments**).
We try to riassume with the following example the concepts seen so far:

```
>>> A = 5
>>> B = 10
>>> A, B = B, A
>>> A
10
>>> B
5
>>> L = [5, 10]
L[1], L[0] = L[0], L[1]
>>> L
[10, 5]
>>> t = [10, 20, 30, 40]
>>> t[:2] = "ciao"
>>> t
['c', 'i', 'a', 'o', 30, 40]
>>> t[:2], t[3] = "prova", 4+5
>>> t
['p', 'r', 'o', 9, 'a', 'a', 'o', 30, 40]
>>> t[:2] = 7
TypeError: can only assign an iterable
>>> t[:2] = 'k'
>>> t
['k', 'o', 9, 'a', 'a', 'o', 30, 40]
>>> p = [44, 66, 77]
>>> q = [55, 88, 99]
>>> p[0], q[1] = 4, 5
>>> p
[4, 66, 77]
>>> q
[55, 5, 99]
```

Example 3.3.7 (**Recap exercises**).
Just as a small recap try to solve the following exercises.

1. Given a **list** `L` of **int**, modify any element, transforming it in its double (*e.g.* given `L = [10, 20, 30, 40]` transform `L` in `[20, 40, 60, 80]`). Write a general program which works *for any* such **list** `L`.

2. Given a **list** `L` of **int**, return `True` if and only if it contains a zero value element. Write a general program also here, utilizing **for** or **while** or both (which is the best?)

Now that we have introduced *mutable* object the concept of *aliasing* becomes more interesting (and maybe more complex).

Remark 3.3.8 (**Aliasing and side effects**).

```
>>> L = [10, 20, 30]
>>> V = L
>>> L[0] = 100
>>> print(V)
[100, 20, 30]
```

Since there is *aliasing* (`V` and `L` are alias for the same object), `V` has been modified by *side effect*. We also say that `L[0] = 100` has the side effect to modify V.

Remark 3.3.9 (**Aliasing and side effects**).
We may have aliasing also on **tuple** (or **int**, or **float** *etc.*; also if for immutable object aliasing is not a true aliasing; check the ids):

```
T = (10, 20, 30)
R = (100, T, 300)
```

The *same* on **list**:

```
L = [10, 20, 30]
W = [100, L, 300]
```

On **list** we may modify `L` (note that we say *modify* and not *re-assign*), and this will be seen also from `W`.

```
>>> L = [10, 20, 30]
>>> W = [100, L, 300]
>>> id(W)
140272586546704
>>> L[1] = 99999
>>> W
[100, [10, 99999, 30], 300]
>>> id(W)
140272586546704
>>> L = [2,3]
>>> W
[100, [10, 99999, 30], 300]
```

Remark 3.3.10 (**Aliasing and side effects: moral**).
Aliasing and *side effect* are present in the language *by design*. They are the *raison d'être* of mutable objects! Yet, they are a delicate, subtle ingredient of the language to be understood, to be controlled; source of errors which are difficult to locate and discover.

Definition 3.3.11 (**Deleting from a 'list': 'del'**).
In order to delete elements from a **list** `L` we can use the following command:

```
del <selection>
# example: del L[i]
```

- Semantics:
  - `<selection>` (`L[i]`) is eliminated from `L`;
  - the elements *following* `<selection>` (`L[i]`) are *shifted to the left* (indexes are recomputed).

Example 3.3.12 (**Deleting from a 'list': 'del'**).
Given a **list** of `i` **int**, delete the zeros:

```
L = [1, 2, 0, 3, 0, 5, 0]
i = 0
while i < len(L):
    if L[i] == 0:
        del L[i]
```

```
    else:
        i = i + 1
# here L is [1, 2, 3, 5]
```

Observe that the index is incremented only when **del** is not executed.
It does not work with **for**:

```
L = [1, 2, 0, 3, 0, 5, 0]
for i in range(len(L)):
    if L[i] == 0:
        del L[i]
# here L is ?
IndexError: list index out of range
```

Don't use **for** on **list** in presence of side effects! Never use **for** on **list** if their length is modified in the **for** body. The heading of the **for** command *freezes* the sequence object (its identity), not its value:

```
L = [1, 2, 0, 3, 0, 5, 0]
for i in range(len(L)):
    if L[i] == 0:
        del L[i]
    print(range(len(L)))

range(0, 7) # range is immutable: frozen in the header
range(0, 7) # another object in comparison to what frozen
range(0, 6)
range(0, 5) # L is mutable: index i becomes > len(L)
range(0, 4)
IndexError: list index out of range
```

For the sake of clarity see also Remark 3.3.22 ahead. We repeat: the heading of the **for** command *freezes* the sequence object (its identity), not its value.

Remark 3.3.13 (**Deleting selection, name**).
In the *modifying* assignment:

```
<selection> = <expression>
```

`<selection>` must be an *explicit selection*.
The same in:

```
del <selection>
```

`<selection>` must be an *explicit selection*.
Compare:

```
>>> L = [10, 20, 30]
>>> del L[1]
>>> L
[10, 30]
```

With:

```
>>> L = [10, 20, 30]
>>> e = L[1]
```

```
>>> e
20
>>> del e
>>> L
[10, 20, 30]
>>> e
NameError: name 'e' is not defined
```

Beware: **del** <name> will remove <name> from the internal state.

Remark 3.3.14 (**Add/Delete elements: slice**).
Considering:

```
<selection> = <expression>
```

<selection> may be a **slice**.
If <selection> is a **slice**:

- RHS must be a **list** (or more precisely an *iterable* which will be converted in a **list**), whose elements are inserted for the slice:

  ```
  >>> t = [10, 20, 30, 40]
  >>> t[: 2] = (1, 2)
  >>> t[: 2]
  [1, 2]
  >>> t
  [1, 2, 30, 40]
  ```

- As a result of the assignment, the **list** at LHS may *grow* or *shrink*.

Example 3.3.15 (**Add/Delete elements: slice**).
Selection may be a **slice**:

```
>>> L = [1, 2, 3]
>>> L
[1, 2, 3]
>>> L[0 : 2] = [10, 20]
>>> L
[10, 20, 3]
>>> L[0 : 1] = [5, 6]
>>> L
[5, 6, 20, 3]
>>> L[-1 :] = [4]
>>> L
[5, 6, 20, 4]
>>> L[1 : 1] = [1, 2]
>>> L
[5, 1, 2, 6, 20, 4]
>>> L[0 : 2] = []
>>> L
[2, 6, 20, 4]
>>> L[0 : 0] = [1]
>>> L
[1, 2, 6, 20, 4]
>>> L[len(L) :] = [30] # the same with L[40: ] = [30]
```

```
>>> L
[1, 2, 6, 20, 4, 30]
```

Remark 3.3.16 (**Slices and 'list': warnings**).
Warnings (given a **list** L):

- **del** L[i : j] is equivalent to L[i : j] = [];
- L[1 : 2] = [4, 5, 6] and L[1] = [4, 5, 6] are two (very!) different things:
  - L[1 : 2] = [4, 5, 6] *extends* L;
  - L[1] = [4, 5, 6] leaves unchanged the lenght of L; the element L[1] is replaced by the (**list**) object [4, 5, 6].

Remark 3.3.17 (**Add/Delete elements: methods**).
Invoking a method on a object of type **list** may *modify* the object.

Definition 3.3.18 (**Some methods semantically equivalent to slicing**).
Let's see some methods on **list** semantically equivalent to slicing. Consider also that, in general, using methods is more efficient.

- L.append(x):
  - append x as last element of L;
  - semantically equivalent to L[**len**(L):] = [x]; (note that in general L[**len**(L)] gives an error).
- L.insert(i, x):
  - insert x in L at index i (shifting indexes), or append if i $\geq$ **len**(L);
  - semantically equivalent to L[i : i] = [x].
- L.extend(V):
  - extends L with **list** V (in place concatenation);
  - semantically equivalent to L[**len**(L):] = V.
- L.clear():
  - empties L;
  - semantically equivalent to **del** L[:].

All of them **return** None (see next Definition).

Now we introduce a new singular **type**.

Definition 3.3.19 (**Type 'NoneType'**).
A new **type** is NoneType.

- Values: only one;
- presentation: None;

- operations: no operations.

`None` is the value that Python uses for those expressions for which the value is not relevant; the shell simply doesn't show `None` as a value; the command `print()`, instead, prints `None`.

```
def prova(s, x):
    s = s + x
    return

>>> %Run prova.py
>>> print(prova(5, 6))
None
```

Example 3.3.20 (**One exercise with methods**).
Using methods, read from the keyboard a series of positive integers, until the user inserts `9999`; store the numbers in a `list`.

Remark 3.3.21 (**Operations vs methods**).
Be careful on this:

```
L = L + [element]
```

It is not the same as:

```
L.append(element)
```

In the first a new object is created; in the second, no new object is created.
Consider the following:

```
>>> L = [10, 20, 30]
>>> W = L
>>> L.append(40)
>>> print(W)
[10, 20, 30, 40]
>>> L = L + [60]
>>> print(W)
[10, 20, 30, 40]
```

Let's return briefly on the theme relative to using `for` on `list`.

Remark 3.3.22 (**'for' with 'list'**).
Don't use `for` on `list` in presence of side effects!
The heading of the `for` command *freezes* the sequence object *i.e.* its *identity*, *not its value*.
Compare:

```
L = [10]
for e in L:
    L.append(e)
print(L)
```

With:

```
L = [10]
for e in L:
    L = L + [e]
print(L)
```

The first is an infinite loop, the second prints `[10, 10]`.

Let's go back with the theme of *assignment* and see one more variant.

Definition 3.3.23 (**Augmented assignment**).
Let's define what is called *augmented* assignment.
Generally, during an iteration scan, often these forms are used:

```
res = res + val
# or also
count = count * val
```

It is possible to use *contracted* forms:

```
res += val
# or also
count *= val
```

A small list of operand relative to *contracted* forms:

```
+=  -=  *=  /=  //=  %=  **= # and even others
```

Operations are overloaded:

```
T += v
```

Can be addition or concatenation *etc.* depending from **type** of `T`. But we have to be more precise:

- `x += v`
- `x = x + v`

These two forms are *similar*, but *not the same*. The first form differs from its *normal* analogue for the following aspects:

1. the operation is done *in place* if possible (mutable objects are modified; no creation of new objects, if possible);
2. `x` is *evaluated only once*;
3. LHS is *evaluated first*.

Meaning that:

1. the operation is done *in place* if possible:

   ```
   L = [1, 2, 3]
   L += [4]
   ```

   And

```
V = [1, 2, 3]
V = V + [4]
```

Are two different things in the sense that += has the meaning of *extend* (`append()`) on **`list`**.

```
>>> L = [1, 2, 3]
>>> id(L)
140350968942832
>>> L += [4]
>>> id(L)
140350968942832

##### vs #####

>>> V = [1, 2, 3]
>>> id(V)
140350968936880
>>> V = V + [4]
>>> id(V)
140350968936560
```

2. `x` is *evaluated only once.*

   Let `f(x)` be a function and consider:

   ```
   L[f(i)] += val
   ```

   And

   ```
   L[f(i)] = L[f(i)] + val
   ```

   The operator += calls `f(i)` only once; if `f(i)` has side effects, they happen once only.

3. LHS is *evaluated first*: LHS is *evaluated first*, then RHS, then the binding in the state is modified. Let `f(x)` be a function:

   ```
   L[f(i)] += exp
   ```

   If `f(i)` has side effects, they happen *before* the evaluation of `exp`.

Definition 3.3.24 (**More methods on 'list'**).
Let's define other methods on **`list`**:

- `L.remove(x)`:
  - remove from `L` the first element `L[i]` equal (==) to `x`; `error` if `x` not in `L`; **`return`** `None`;
- `L.copy()`:
  - **`return`** a (*shallow*) copy of `L`; equivalent to `L[:]` in slice notation.

```
>>> L = [1, 2, 3]
>>> id(L)
140350968937120
>>> M = L.copy()
>>> id(M)
140350968943072
>>> F = L[:]
>>> id(F)
140350968938400
```

- `L.reverse()`:
  - reverses `L` (in place); **`return`** `None`; What is the difference with `L = L[: : −1]`? Remember the mantra: 'operations create new objects'.

```
>>> L = [1, 2, 3]
>>> id(L)
140350968936080
>>> L.reverse()
>>> id(L)
140350968936080
>>> L
[3, 2, 1]
>>> L.reverse()
>>> L
[1, 2, 3]
>>> L = L[: : −1]
>>> L
[3, 2, 1]
>>> id(L)
140350968946352
```

- `L.sort()`:
  - sort `L` (in place);
- `L.pop(i)`:
  - **`return`** and removes the element at index `i` *i.e.* **`return`** the value `L[i]` and modify (in place) the **`list`**, removing the element at index `i` leaving unchanged the identity of `L`; `pop()` **`return`** and removes `L[−1]`.

Now let's analyze this question: can functions modify global mutable object?

Remark 3.3.25 (**Functions can't modify global immutable objects**). Responding to the question if functions can modify global *mutable* objects, we observe that with *immutable* objects this is impossible:

```
b = 20 # remember that the only way to 'modify' an
def f(a): # immutable object is through name assignment
    a = 10 # and names inside functions are local to
    b = 100 # functions (we can't 'modify' immutables but
    return None # assign new objects at the same name)
```

```
f(b)
print(b)
>>> 20
```

a, b in the body of the function f(a) are local names.
Consider also the followings in order to understand the roles of the values passed to the function and the local variables:

```
def f(a):
    a = 10
    b = 100
    return a
# function is running
>>> f(5)
10

###### NEW SHELL SESSION ######

b = 5
def f(a):
    a = a == 5
    return a
# function is running
>>> f(b)
True
>>> f(6)
False

###### NEW SHELL SESSION ######
b = 5
def f(a):
    a = 10
    a = a == 5
    return a
# function is running
>>> f(b)
False
```

Definition 3.3.26 (Global declaration).
So, in general, functions can't modify global *immutable* objects, unless a *global declaration* (**global**) is used.

```
b = 20
def f(a): # the same with def f()
    global b
    a = 10
    b = 100
    return None
# function is running
>>> f(b)
>>> print(b)
100
>>> print(a)
NameError: name 'a' is not defined
```

a is *local*, b is the one in the main frame.

Remark 3.3.27 (**Functions can modify global mutable objects**).
But functions can modify global *mutable* objects and this is possible in several ways:

1. through a mutable actual parameter:

```
b = [20]
def f(a):
    a[0] = 100
    return None
# function is running
>>> f(b)
>>> print(b)
[100]
```

   Of course the assignment `a[0] = 100` would result in an error when `f` is called on an *immutable* actual parameter;

2. through a *global name*, bound to a `mutable` object:

```
b = [20]
def f():
    b[0] = 100
    return None
# function is running
>>> f()
>>> print(b)
[100]
```

Observe: at the left of = (inside the body of the function) there is a *selection* not a *name*; hence the rule: 'name at the left of = is local' cannot be applied. See:

```
>>> b = [20]
>>> def f():
        b = 7
        return None
# function is running
>>> f()
>>> print(b)
[20]
>>> def g():
        b = [100]
        return None
# function is running
>>> g()
>>> b
[20]
>>> def h(b):
        b = [100]
        return None
# function is running
>>> h(b)
>>> b
[20]
```

We usually form **list** out of other sequences, *e.g.* if we want the doubles of the number from 0 to 100 we can write the following code:

```
doubles = []
for i in range(0, 101):
    doubles.append(2 * i)
```

It would be nice having a *compact way* to express this, something like:

```
doubles = the list of 2 * i for i in range(0, 101)
```

Or also if we want the even numbers out of a given list of integers LN:

```
LN = ...
even = []
for i in LN:
    if i % 2 == 0:
        even.append(i)
```

It would be nice having a *compact way* to express this, something like:

```
even = the list of those i in LN if i % 2 == 0
```

Python allows this form of expression which is called comprehension in set-theory, in math: the comprehension principle is the operation by which, given a condition expressible by a formula $\varphi(x)$, we form the set of all those x meeting the condition: $\{ x \mid \varphi(x) \}$.

Remark 3.3.28 (**Comprehension and its unwinded form**).
A **list** comprehension is always equivalent to its *unwinded* form, meaning for example that:

```
even = []
for i in range(0, 101):
    if i % 2 == 0:
        even.append(i)
```

It is equivalent to (correct Python code):

```
even = [i for i in range(0, 101) if i % 2 == 0]
```

It is not difficult to imagine that we can also have *nested* comprehension. Before giving the general definition we propose some examples in order to familiarize with the formal form.

Example 3.3.29 (**Some comprehensions**).
Examples:

1. the **list** of numbers from 0 to i, for i less than 20:

   ```
   >>> [[k for k in range(i)] for i in range(21)]
   # output
   [[], [0], [0, 1], [0, 1, 2], [0, 1, 2, 3], ...]
   ```

   It is equivalent to:

```
LL = []
for i in range(21):
    AL = []
    for k in range(i):
        AL.append(k)
    LL.append(AL)
```

2. These two forms are equivalent:

```
>>> T = [(x, y) for x in range(2) for y in range(3)
          if x < y]
>>> T
[(0, 1), (0, 2), (1, 2)]
```

It is equivalent to:

```
T = []
for x in range(2):
    for y in range(3):
        if x < y:
            T.append((x, y))
```

3. The following gives to us the approximations of mathematical π from zero to five decimal numbers:

```
>>> from math import pi
>>> P = [round(pi, i) for i in range(6)]
>>> P
[3.0, 3.1, 3.14, 3.142, 3.1416, 3.14159]
```

Now is the right moment to give the general definition which is a bit obscure.

Definition 3.3.30 (**Comprehension**).
Basically we have two forms:

1. a simple version:

```
[expression(name) for name in sequence]
```

Where `expression` (note that `expression` depends on `name`) in a comprehension is always evaluated (see on *pythontutor*):

- after the `for name in sequence`;
- in its local frame: `name` is *local* to the comprehension.

2. A more complex version:

```
[expression for name in sequence series of for/if]
```

Equivalent to:

```
res = []
for name in sequence:
    series of for/if:
        res.append(expression)
```

Example 3.3.31 (**Series of 'for'/'if' in a comprehension**).
To understand how to combine correctly the series of **for/if** in a comprehension imagine always the equivalence of the comprehension itself with its iterative *unwided* form.
The next two comprehensions are equivalent:

```
>>> T = [(x, y) for x in range(3) for y in range (5) if x
         % 2 == 0 if y % 2 != 0]
>>> T
[(0, 1), (0, 3), (2, 1), (2, 3)]
################################
>>> H = [(x, y) for x in range(3) if x % 2 == 0 for y in
         range (5) if y % 2 != 0]
>>> H
[(0, 1), (0, 3), (2, 1), (2, 3)]
```

But pay attention to the order:

```
G = [(x, y) for y in range(5) if y % 2 != 0 for x in
     range(3) if x % 2 == 0]
>>> G
[(0, 1), (2, 1), (0, 3), (2, 3)]
```

And obviously to the correctness:

```
G = [(x, y) for x in range(3) if y % 2 != 0 for y in
     range(5) if x % 2 == 0]
UnboundLocalError: local variable 'y' referenced before
assignment
```

Now, in order to give a practical view of what is possible doing with **list** and more generally with comprehensions, we propose some tasks and exercises to do (or just to read in order to understand the power of **list** and comprehensions).

Example 3.3.32 (**Possible exercises on comprehension**).
Exercises:

1. given a unary function (meaning that it takes one parameter) `f` and a sequence `S`, return the **list** of the application of `f` to the elements of `S`.
2. Given two sequences `S1` and `S2` return the **list** of the elements present in both `S1` and `S2` (don't bother about multiplicity: if `e` appears in both `S1` and `S2`, maybe many times in `S1`, or in `S2`, or in both, it may appear with whatever multiplicity in the result).
3. Given an integer k, list Pythagorean triples: (a, b, c) such that $a^2 + b^2 = c^2$, for $a + b + c \leq k$.

Example 3.3.33 (**Permutations via comprehension**).
Exercise: define a function **def** `perm(L)` which **return** the **list** of all the permutations of the elements of `L`; do not modify `L`.
Example:

```
perm([1, 2, 3])
# return
[[1, 2, 3], [1, 3, 2], [2, 1, 3], ..., [3, 2, 1]]
```

Hint: argue recursively; the permutations of `L` are obtained by taking as first element, any element of `L`, and taking then all the permutations of ...; you need an auxiliary function `def erase(e, L)` which `return` a copy of `L` from which `e` has been removed.

Example 3.3.34 (**Ordered insertion**).
Exercise: write a function `def ins_ord(L, el)` which given an ordered `list` of integers `L` and an integer `el`, insert (in place) `el` in `L` in increasing order.
Example:

```
>>> S = [2, 4, 6, 8]
ins_ord(S, 5)
# must modify S in
>>> S
[2, 4, 5, 6, 8]
```

Sometimes we need to iterate *inside* another iteration. This programming schema is called *double* (or *nested*) iteration.

Example 3.3.35 (**Double/Nested iteration**).
A trivial example that can be implemented usign double iteration: given a `tuple` of `tuples` of integers, count the zeros.
Example:

```
def zeros(ToT):
    count = 0
    for t in ToT:
      for e in t:
          if e == 0:
              count += 1
    return count
```

Example 3.3.36 (**Floyd triangles**).
A *Floyd triangle of order* n, with $n \geq 1$ is a mathematical object of n rows with the integers from 1; any row has one more element of the previous row:

```
# A Floyd triangle of order 5
1
2  3
4  5  6
7  8  9  10
11 12 13 14 15
```

As exercise write a function `Floyd(n)` which prints a Floyd triangle of order `n`. We want a *pretty print* (see next Definition).

Definition 3.3.37 (**Pretty print**).
In Python `print()` may have additional arguments *by keyword*:

- `sep`: string used to *separate* values in the same `print()` (default: `' '`);
- `end`: string used to *terminate*, after a `print()` (default: *'newline'* `\n`[2]).

[2]While `\t` stands for *'tab'* in Python. Look in the official documentation for other.

Example:

```
def prova_print():
    print(10, 20)
    print(30, 40)
    print(10, 20, sep = ' & ', end = ' * ')
    print(30, 40, sep = ' & ')
    print(50)
    return

>>> prova_print()
10 20
30 40
10 & 20 * 30 & 40
50
```

Definition 3.3.38 (**Double iteration: matrices**).
Python does not have a **type** for array/matrices (but we shall see the library `NumPy`).
We may encode an $n \times m$ matrix as:

- **list** of `lists`:
  - a **list** of n elements (the rows);
  - each element is a **list** of m elements (the m elements of the row *i.e.* the columns).

```
LA = [ [0, 1],
       [3, 2],
       [5, 6] ]
```

Correspond mathematically to:

$$A = \begin{pmatrix} 0 & 1 \\ 3 & 2 \\ 5 & 6 \end{pmatrix}$$

Where:

- element a_{ij} of matrix A is encoded as element `L[i−1][j−1]` of **list** `LA` (supposing that the indexes of A start at 1). For simplicity: suppose that both start from zero.

Also for this topic we propose some useful exercises.

Example 3.3.39 (**Exercises on matrices**).
Exercises:

1. write a function **def** `is_sum_mat(A, B, S)` which given matrices `A`, `B`, `S` encoded as **list** *verify* that `S` is the matrix sum of `A` and `B` where for any `i` and `j` we have that: `S[i][j] = A[i][j] + B[i][j]`.

2. Write a function **def** `sum_mat(A, B)` which given matrices `A`, `B` encoded as **list** **return** a *new* **list** `S`, the matrix sum of `A` and `B`, as a **list** of `lists`.

3. Write a function which computes and **return** the *transpose* of a matrix, taken as a parameter.

4. Write a function which takes two **list** (not matrices) of the same lenght, and computes and **return** the *scalar product* of the two **list**.

5. Write a function which **return** the *product* of two matrices `A` and `B`, given as parameters (you may assume they are in the correct format).

   Hint: you may use the transpose to make the scalar product between the rows of `A` and the rows of the transpose of `B`.

Now we are ready to introduce another type.

Remark 3.3.40 (**Dictionaries: generalitazion of 'list'**).
A **list** is a (mutable) mapping between *indexes* and objects, where for *indexes* we mean initial segment of natural numbers.
A `dictionary` is a generalization: a (mutable) mapping between *immutable objects* and objects.

Example 3.3.41 (**A paradigmatic problem: frequency count**).
Given the string:

```
LC = 'When in the Course of human events'
```

Count the frequency of each letter, including spaces ('W' appears 1 time, 'h' appears 3 times, ' ' appears 6 times *etc.*).
As a first try use **list**.
But we notice that it would be much easier if we could directly use the elements of `LC` (characters of a string) as indexes of objects (integer counts) in a suitable sequence/container.

Definition 3.3.42 (**Dictionaries: mapping**).
A new **type** is `dictionary` (**type dict**).
Characteristics:

- compound **type**, *mutable*;
- it is a finite mapping between:
  - *immutable* objects: the *keys* of the dictionary;
  - arbitrary objects: the *values* of the dictionary.
- Generalizes **list**: a **list** `S` is a mapping between `[0 : len(S)]` and objects;
- values: finite mapping between ...;
- presentation:
  - `{}` is the empty **dict**;
  - `{k_1 : ob_1, k_2 : ob_2, ..., k_n : ob_n}` the mapping sending `k_i` into `ob_i`; the pairs `k_i : ob_i` are the *items* of the dictionary.

Example 3.3.43 (**Dictionaries: mapping**).
Example:

```
{1 : 10, 2 : 20, 'one' : 20}
```

Keys:

- all distinct;
- all of immutable type (also any component, if any, should be immutable; meaning that if we have a key of immutable type, but compound, which has a component of mutable type, then we raise an error; see next Example).

Remark 3.3.44 (**Dictionaries main operations**).
Main operations related to dictionaries (suppose to have a `dict D`):

- lenght: `len(D) return` number of items (pair);
- selection: `D[k]` where `k` is a key (`error` if `k` is not a key);
- deletion: `del D[k]` where `k` is a key;
- mutable: hence selection may be LHS of assignment;

  ```
  D[k] = object
  ```

  Set in `D` the pair `k : object`, creating it if necessary;
- `k in D`: `True` if and only if `k` is a key in `D`.

Example 3.3.45 (**Dictionaries: some common syntax error**).
Which of the following raise an error?

```
>>> D1 = {1 : 10, 2 : 20}
>>> D2 = {1 : 10, 'a' : 3.14, 3 : 3}
>>> D2['a'] = D2[1]
>>> D2
{1: 10, 'a': 10, 3: 3}
>>> D3 = {1 : [1, 2, 3], 3 : {1 : 10, 2 : 20, 3 : 30},
          2 : (1, 2, 3)}
>>> D3[5] = 'pippo'
>>> D3[1][0] = 100
>>> D3[2][1] = 20     # Error: tuple is not modifiable
TypeError: 'tuple' object does not support item
assignment
>>> D4 = {(1, 2) : [1, 2]}
>>> D5 = {[1, 2]: (1, 2)}   # Error: key is modifiable
TypeError: unhashable type: 'list'
>>> D6 = {(1, [2]) : 23}    # Error: key is modifiable
TypeError: unhashable type: 'list'
```

Remark 3.3.46 (**Some aspects to consider**).
Regarding dictionaries consider the followings:

- no slices;
- no concatenation, they are *not sequences*;

- `D1 == D2` this is `True` if and only if `D1` and `D2` have the same items, *indipendently from the order of the items*;
- listing the pairs of a dictionary respects the insertion order (only from Python 3.7).

Remark 3.3.47 (**Dictionaries vs 'list' of pairs**).
Consider the followings:

```
D = {1 : 10, 'a' : 3.14, 3 : 30, 5 : 50}
###
L = [(1, 10), ('a', 3.14), (3, 30), (5, 50)]
```

They have:

- same information;
- different efficiency of the access operation *by key*:
  - **`list`**: sequential, hence `O(len(L))` time;
  - **`dict`**: hash[3] access, constant time;
- access *by key* to a **`dict`** is as efficient as an access *by index* to a *list*.

Remark 3.3.48 (**Iteration order on 'dict'**).
Be `D` a dictionary. Consider:

```
for e in D:
    <body>
```

`e` varies on the *keys* of `D`, but in which order?

- From Python 3.7: in the order in which keys have been inserted in `D`;
- before: in an arbitrary order depending on the machine.

Remark 3.3.49 (**Iteration on 'dict': side effects**).
Similar to **`list`**, don't use **`for`** on **`dict`** in presence of side effects. Never (never!) use **`for`** on dictionaries, *if their lenght is modified in the* **`for`** *body.*
Notice that this time it is an *absolute* prohibition: *a modification to a* **`dict`** *which is used to control a* **`for`** *raises a run time error.*

Definition 3.3.50 (**The method 'get()'**).
Let's remember the frequency count on a sequence `S`:

```
# Common situation: check if key present, otherwise
# update
Freq = {}
for e in S:
    if e in Freq:
        Freq[e] += 1
    else:
        Freq[e] = 1
```

[3] An hash function is any function that can be used to map data of arbitrary size to fixed-size values.

```
# Notice that instead the following code is wrong
# it raises a key error
# WRONG !
Freq = {}
for e in S:
    Freq[e] += 1
```

Now consider the following method on a dictionay `D`:

- `D.get(key, default)`:
  - **return** `D[key]` if it exists; otherwise **return** `default`.

With this method we can perform the frequency count saving code:

```
Freq = {}
for e in S:
    Freq[e] = Freq.get(e, 0) + 1
```

Definition 3.3.51 (**Dictionaries: 'view object' methods**).
Let's define other methods related to dictionaries. Let `D` be a **dict**:

- `D.keys()`;
- `D.values()`;
- `D.items()`.

They **return** *view objects* (*dynamic* **list**), on keys, values, items respectively (see deeply the meaning of *view object* in a shell). Examples:

```
>>> D = {'a': 333, 1 : [5,6]}
>>> D.keys()
dict_keys(['a', 1])
>>> C = D.keys()
>>> C
dict_keys(['a', 1])
>>> type(C)
<class 'dict_keys'>
D.values()
dict_values([333, [5, 6]])
>>> G = D.values()
>>> G
dict_values([333, [5, 6]])
>>> D['a'] = 'prova'
>>> D.values()
dict_values(['prova', [5, 6]])
>>> G
dict_values(['prova', [5, 6]])
```

On these views the operator **in** is defined.
We may *freeze* these dynamic views:

```
L = list(D.keys()) # or
T = tuple(D.values()) # etc.
```

Definition 3.3.52 (**Dictionaries: a substitute for concatenation**).
Concatenation does not exist for dictionaries.
The method `update()` does something similar, when possible.
Let's `D1` and `D2` two **`dict`**:

- `D1.update(D2)`:
  - extends `D1` with the items of `D2`; if there are *equal keys* in `D1` and `D2`, the ones in `D2` (*update*) are chosen:

```
>>> D1 = {'house' : 'casa', 'cat' : 'gatto',
          'red' : 'rosso'}
>>> D2 = {'cat' : 'micio', 'grey' : 'grigio'}
>>> D1.update(D2)
>>> print(D1)
{'house': 'casa', 'cat': 'micio', 'red': 'rosso',
 'grey': 'grigio'}
```

Remark 3.3.53 (**From 'dict' to 'list'**).
Let `D` be a dictionary. If we would like to pass from **`dict`** to **`list`** we can:

1. use methods:
   - `D.keys()`;
   - `D.values()`;
   - `D.items()`.
2. Freeze them into **`list`**. In particular:
   - **`list`**`(D.items())`;
     - **`return`** as **`list`** of `tuples` the items of the dictionary.

   Example:

```
>>> list({1 : 'uno', 2 : 'due', 3 : 'tre'}.items())
[(1, 'uno'), (2, 'due'), (3, 'tre')]

#### But also it is valid:
>>> tuple({1 : 'uno', 2 : 'due', 3 : 'tre'}.items())
((1, 'uno'), (2, 'due'), (3, 'tre'))
```

Remark 3.3.54 (**From 'dict' to 'list': warning equivalence**).
Let `D` be a dictionary. Warning:

```
list(D)
```

Is equivalent to:

```
list(D.keys())
```

Remark 3.3.55 (**'dict()' on a list of pairs**).
`dict``()` on a *list of pairs* **`return`** a **`dict`**:

```
>>> dict([(1, 'uno'), (2, 'due'), (3, 'tre')])
{1: 'uno', 2: 'due', 3: 'tre'}
>>> dict([[1, 'uno'], [2, 'due'], [3, 'tre']])
{1: 'uno', 2: 'due', 3: 'tre'}
>>> dict(((1, 'uno'), (2, 'due')))
{1: 'uno', 2: 'due'}
>>> dict(([1, 'uno'], (2, 'due')))
{1: 'uno', 2: 'due'}
```

Definition 3.3.56 (**Building a list of pairs: 'zip()'**).
Be `L1` and `L2` two lists (not necessary a **`list`**, can be also a **`tuple`** or a **`str`**). In order to build a *list of pairs* is handy the function:

- **`zip`**`(L1, L2)`:
  - **`return`** a special object **`zip`** (an *iterator*) which codifies a *list of pairs*;
  - if one list is shorter, we take the shortest as reference;
  - careful: one **`zip`** object may be used only once (more on this later).

```
# Note
>>> L1 = (1, 2, 3)
>>> L2 = 'lista'
>>> zip(L1, L2)
<zip object at 0x7fecd46a9640>
>>> dict(zip(L1, L2))
{1: 'l', 2: 'i', 3: 's'}
>>> list(zip(L1, L2))
[(1, 'l'), (2, 'i'), (3, 's')]
>>> tuple(zip(L1, L2))
((1, 'l'), (2, 'i'), (3, 's'))
>>> for x, y in zip(L1, L2):
      print((x, y))

(1, 'l')
(2, 'i')
(3, 's')
>>> L3 = 'prova'
>>> for x, y, z in zip(L1, L2, L3):
      print((x, y, z))

(1, 'l', 'p')
(2, 'i', 'r')
(3, 's', 'o')
>>> for x, y in dict(zip(L1, L2)):
      print((x, y))

TypeError: cannot unpack non-iterable int object

###############

>>> dishes = ['pizza', 'paella', 'hamburger']
>>> countries = ['Italy', 'Spain', 'USA']
```

```
>>> zip(countries, dishes)
<zip object at 0x7ff6b2e9df00>
>>> special = zip(countries, dishes)
>>> D_special = dict(special)
>>> D_special
{'Italy': 'pizza', 'Spain': 'paella', 'USA': 'hamburger'}

# Shortest as reference
>>> dishes = ['pizza', 'paella']
>>> countries = ['Italy', 'Spain', 'USA']
>>> special = zip(countries, dishes)
>>> D_special = dict(special)
>>> D_special
{'Italy': 'pizza', 'Spain': 'paella'}

# Used only once
dishes = ['pizza','paella','hamburger']
>>> countries = ['Italy','Spain','USA']
>>> zip(countries, dishes)
<zip object at 0x7f8e0e11c140>
>>> special = zip(countries, dishes)
>>> D_special = dict(special)
>>> D_special
{'Italy': 'pizza', 'Spain': 'paella', 'USA': 'hamburger'}
>>> E_special = dict(special)
>>> E_special
{}

# But notice the differences
>>> dishes = ['pizza','paella','hamburger']
>>> countries = ['Italy','Spain','USA']
>>> zip(countries, dishes)
<zip object at 0x7f8e0e11a1e0>
>>> special = zip(countries, dishes)
>>> special
<zip object at 0x7f8e0e118460>
>>> D_special = dict(special)
>>> D_special
{'Italy': 'pizza', 'Spain': 'paella', 'USA': 'hamburger'}
>>> E_special = dict(special)
>>> E_special
{}
>>> E_special = dict(zip(countries, dishes))
>>> E_special
{'Italy': 'pizza', 'Spain': 'paella', 'USA': 'hamburger'}
>>> G_special = dict(zip(countries, dishes))
>>> G_special
{'Italy': 'pizza', 'Spain': 'paella', 'USA': 'hamburger'}
>>> dict(zip(countries, dishes))
{'Italy': 'pizza', 'Spain': 'paella', 'USA': 'hamburger'}
>>> dict(zip(countries, dishes))
{'Italy': 'pizza', 'Spain': 'paella', 'USA': 'hamburger'}
```

The comprehension principle and implementation is valid for dictionaries too.

Definition 3.3.57 (**Comprehension on dictionaries**).
On **dict**, comprehension has the following syntax:

```
{key(name) : val(name) for name in sequence}
```

Examples:

```
{i : i ** 2 for i in range(10)}
{i : i ** 2 for i in range(10) if i % 2 == 0}
{i : [k for k in range(i)] for i in range(10)}
```

It is possible to apply the comprehension principle also on another Python type which we don't cover in this course *i.e.* **set**; just we give an high level definition of this type.

Definition 3.3.58 (**Sets**).
set are *mutable* collection of objects without repetition and unordered; they are iterable.

But it is not possible to apply comprehension on everything.

Remark 3.3.59 (**No comprehension on 'tuple'**).
We highlight that there is no comprehension on **tuple**; yet expressions which *seem* '**tuple** comprehension' are legal Python code, like:

```
H = (i * 2 for i in range(10))
```

Definition 3.3.60 (**Generators: super simplified view**).
Consider:

```
G = (i * 2 for i in range(3))
```

A `generator` is a kind of '*potential* sequence/**tuple**' which is able to *generate* the elements of the sequence, through the predefined function **next()**:

1. **next**(G): evaluates to 0;
2. **next**(G): evaluates to 2;
3. **next**(G): evaluates to 4;
4. **next**(G): the generator is exhausted, an error is raised.

Some important aspects:

- an important use: a generator may be the <sequence> used in a **for**:

  ```
  for e in G:
      print(e)
  ```

  here **for** calls **next**(G) at any iteration (and assign the result to e). A generator is a particular case of an `iterator`, the general structure on which **next()** is defined and on which we may iterate with a **for**.

- We have seen that **list** comprehension is a shorthand for an *explicit iteration* involving `append()`; is there an *explicit version* of a generator? Yes but we need to introduce something new (see also next Definition):

```
G = (i for i in range(10))

# is equivalent to
def genstup(n):
    for i in range(n):
        yield i
G = genstup(10)
```

Also:

```
Q = (i ** 2 for i in range(10) if i % 2 == 0)

# is equivalent to
def quadperf(n):
    for i in range(n):
        if i % 2 == 0:
            yield i ** 2
Q = quadperf(10)
```

Definition 3.3.61 (**Command 'yield'**).
The command:

```
yield <expression>
```

- Could be used only inside the body of a function. The presence of `yield` make the `def` define a generator;
- returns the value of `<expression>` and *suspends the evaluation of the function* (the frame is not destroyed, all the values of the local names are preserved).

In order to highlight the main differences between a classical Python function and a generator we recall briefly the most important aspects of a (classical) function.

Remark 3.3.62 (**Recall: function**).
Functions have two linguistic appearences:

1. definition:

```
def name(<formal parameters>):
    <body>
```

2. Use/Call:

```
name(<actual parameters>)
```

   (a) Evaluate the `<actual parameters>`;
   (b) bind these values to the `<formal parameters>`;
   (c) evaluate the `<body>` of `name` until a `return` is found;
   (d) evaluation happens in local frame, which is destroyed at `return`.

And now we present a more general definition regarding generators in order to complete the concept (also considering the differences with classical functions).

Definition 3.3.63 (**Generators: a peculiar form of functions**).
Similar to functions, generators have two linguistic appearences:

1. definition:
   - any **def** whose body contains (at least) one `yield`.
2. Use/Call:
   (a) a call to the name of the generator (together with its parameters) returns a generator object (the body is *not* evaluated yet);
   (b) successive calls to the function **next**`(G)` (we suppose here that we have assigned the generator object to the name `G`) on the generator object cause the execution of the body of the generator, untill a `yield` is found;
   (c) `yield <expression>` suspends the execution of the body, returns the value of `<expression>` as *value* of the call to **next**`(G)`; *the frame of the call is not destroyed and it is resumed at the next call* to **next**`(G)`.

We invite to observe a stream of the execution flow related to generators on *pythontutor*; for example observe the stream related to the following.

Example 3.3.64 (**A generator stream**).
Let's give a generic generator stream as example:

```
def quadperf(n):
    for i in range(n):
        if i % 2 == 0:
            yield i ** 2
Q = quadperf(5)
# function is running
>>> next(Q)
0
>>> next(Q)
4
>>> next(Q)
16
>>> next(Q)
Traceback (most recent call last):
File "<pyshell>", line 1, in <module>
StopIteration
```

3.4 Recursion

Functions may call functions in their body. Hence it makes sense to introduce the following.

Definition 3.4.1 (**Recursive function**).
A function may call *itself* in its body. In this case we say that a function is *recursive*.

Example 3.4.2 (**Recursion: a form of repetition**).
Let's see an example of recursion.

```
# print 'OK' n times, then print 'done!'
def printrec(n):
    if n == 0:
        print('done!')
    else:
        print('OK')
        printrec(n - 1)
```

Example 3.4.3 (**Recursion: factorial**).
As other example we propose the classical factorial definition:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n(n-1)! & \text{se } n \geq 1 \end{cases}$$

e.g.

$$3! = 3 \cdot 2! = 3 \cdot (2 \cdot 1!) = 3 \cdot (2 \cdot (1 \cdot 0!)) = 3 \cdot (2 \cdot (1 \cdot 1)) = 6$$

In Python we can define it as:

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

We suggest to take trace, as example, of `fact(3)` on *pythontutor*: we will see *multiple active frames* on the stack and *multiple instances of the local names*.

Remark 3.4.4 (**Recursion: ingredients**).
We can build a *recursion* in this way:

- essentially we can consider two cases:
  - *base* case;
  - *recursive* case;
- in order to reach a *termination*:
  - the recursive case *recurs* on a *simpler* case, where with *simpler* we mean closer to the base case (whatever this means).

Example 3.4.5 (**Palindromic exercise**).
As a possible application of a recursive function is to verify if a string is palindromic. Try!

Let's introduce a general theorem of the computation theory.

Theorem 3.4.6 (**Equivalence between recursion and iteration**).
Any computation that is expressible through iteration is also expressible through recursion and vice versa.

As conclusion for this paragraph we propose some exercises on recursion (and iteration).

Example 3.4.7 (Exercises on recursion and iteration).
Write recursive functions for the following problems. Also, write an iterative solution.

1. Sum of a sequence of integers.

2. Linear search: given an unordered sequence `S` and an object `el`, **`return`** `True` if and only if `el` is an element of `S`.

   Hint: write a recursive auxiliar function `ric_lin_aux(S, index, el)` which **`return`** `True` if and only if `el` is element of `S[index :]`; then the real function `ric_lin(S, el)` is simply an initial call to `ric_lin_aux(S, 0, el)`.

3. Your recursive solution of 2. probably uses slices, that are *copies* of portion of `S`. Write a completely *in place* version (no slices).

Chapter 4

Object oriented programming

4.1 Classes and objects

It is the right moment to discuss properly about the living hearts of an *object oriented programming language i.e.* `classes` and `objects`.

Remark 4.1.1 (**A brief recap**).
We already know that:

- **`object`**:
  - *value* incapsulated into an *identity*;
  - *methods* on the *capsule*;
  - available *methods* depend on the **`type`**.
- `Type` (`Class`):
  - *collection* of objects which share:
    - structure;
    - operations;
    - methods.

Remark 4.1.2 (**Viewing from the class**).
From the point of view of the **`class`**:

- a **`class`** (**`type`**), define:
  - the structure of its objects;
  - the available operations and methods.
- The `objects` of the **`class`**:
  - are the *values* of the class, the `instances`;
  - they *share* the structure and the methods defined in the class.

Example 4.1.3 (**Instances of a class**).
So `(1, 2, 3)` is an `instance` (value) of **`tuple`** and `[1, 2, 3]` is an `instance` of **`list`**.

Python is a democratic language.

Remark 4.1.4 (**User may define**).
Classes and instances may be defined by the user.
User-defined classes and their instances are first-class citizens, same rights of any other value!
We may bind them to names, insert them in other objects, pass them to functions, functions may return them.

Let's give now a generic definition.

Definition 4.1.5 (**Classes and objects**).
We can't define `class` without referring to `object` and viceversa:

- `class`: a way for aggregating similar data togheter with the operations which manipulate them;
- `objects` which share the structure defined by a class (and the operations defined in the class) are the `instances` of the class.

Example 4.1.6 (**Abstracting cartesian points**).
A `class` for cartesian points:

- a `point`:
  - two informations: `x` and `y`;
  - any point is created at the origin $(0, 0)$.
- Two `operations`/`methods`:
  - `whoareyou()` `return` the pair (x, y);
  - `move(delta)` modifies the point from (x, y) to $(x + \delta, y + \delta)$.

Let's create with Python code:

```
class Point:
    def __init__(self):
        self.x = 0
        self.y = 0
    def whoareyou(self):
        return self.x, self.y
    def move(self, delta):
        self.x += delta
        self.y += delta
```

Things to be discussed:

- instances?
- Who is this `self` parameter?
- The funny double underscore of method `__init__`?

Let's investigate deeply the three open questions mentioned in the last Example.

Remark 4.1.7 (**Creating instances**).
Consider implemented this portion of code:

```
class Point:
    def __init__(self):
        self.x = 0
        self.y = 0
    def whoareyou(self):
        return self.x, self.y
    def move(self, delta):
        self.x += delta
        self.y += delta
```

In order to create `instances`:

- use the name of the `class` as a function;
- it `return` an `instance`;
- its structure is defined in the method `__init__`.

Example:

```
p = Point()
>>> p
<__main__.Point object at 0x7fae2d6a8a10>
Point()
<__main__.Point object at 0x7fae2d6a8d50>
Point()
<__main__.Point object at 0x7fae2d6a8b90>
```

We may access the `attributes` of (the data stored in) an instance:

- use the `dot` notation:

```
>>> p.x
0
>>> print(p.y)
0
```

`Attributes` are *mutable*:

```
>>> p.x += 1
>>> print(p.x)
1
>>> p.x -= 1
>>> p.x
0
```

`Methods` may be *called on* instances:

```
>>> print(p.whoareyou())
(0, 0)
>>> type(p.whoareyou())
<class 'tuple'>
>>> p.move(2)
>>> print(p.whoareyou())
(2, 2)
```

Remark 4.1.8 (**The first parameter: 'self'**).
Consider implemented this portion of code:

```
class Point:
    def __init__(self):
        self.x = 0
        self.y = 0
    def whoareyou(self):
        return self.x, self.y
    def move(self, delta):
        self.x += delta
        self.y += delta
```

- The `dot` notation ensures that:
  - the first parameter of a method, when called, is a reference to the object receiving the method;

    ```
    >>> p = Point()
    >>> p.whoareyou()
    ```

  - The first parameter of a method is bound to the receiver.
- *By convention*, as first parameter we use **`self`** (but truly we could use any other word *e.g.* `ddd`);
- **`self`** must be used to access the attributes of the instance.

Definition 4.1.9 (**The 'init' method**).
The `__init__` method:

- is called behind the scenes when an *instance is created* (*through the name of the class*);

  ```
  class Point:
      def __init__(self):
          self.x = 0
          self.y = 0
  ##########################
  p = Point()
  ```

- We use it to *initialize* the instances of the class:

  ```
  class Point:
      def __init__(self):
          self.x = 2
          self.y = 8
  ```

- May have *additional parameters*, besides **`self`**; when an istance is created we must provide values for those parameters:

```
class Point:
    def __init__(self, xx, yy):
        self.x = xx
        self.y = yy
##################################
p = Point(0, 1)
```

- It is used for the *creation of attributes*:

```
class Point:
    def __init__(self):
        self.x = 0
        self.y = 0
```

  Observe that:

  - in `self.x = 0` the `self.x` on the left of `=` creates an *instance attribute*;
  - `self.x` is a kind of generalised name, hence when at the left of `=`, creates new bindings.

Definition 4.1.10 (**The 'magic dunder'**).
The *double underscore* `__` is pronounced '*dunder*'.
Methods of the form `__name__` are '*magic methods*', *by convention* reserved for special purposes by the Python machine; we should not fiddle with them, besides their intended purpose. We will see many of them.

Definition 4.1.11 (**The command 'class'**).
The command `class` introduces a class definition:

```
class <name>:
    <commands>
    <methods definition>
```

Both `<commands>` and `<methods definition>` are optional. The simplest class definition (with no structure whatsoever):

```
class A:
    pass
```

We alredy mentioned that the class `<name>` may be *called as a function*:

- it creates an instance of the class *calling implicitly* the method `__init__`; the `<name>` call should have as many arguments as the ones of `__init__`.

Definition 4.1.12 (**Method definitions**).
Method definitions are:

- like function definitions (well: syntactically they *are* function definitions);
- a method *acts on the objects receiving it*;
- a method has always *at least one parameter*:

  - the first parameter of a method (which we call **self** by convention) is *bound by the dot notation to the receiving object.*

Definition 4.1.13 (**Call: method as a function**).
To *call* a method *as a function*: fully qualify *its name* with the *class name*. You must provide the *instance as first argument*:

```
class Point:
    def __init__(self, xx, yy):
        self.x = xx
        self.y = yy
    def move(self, delta):
        self.x += delta
        self.y += delta
p = Point(0, 0)
##################################
p.move(3)
# equivalent to
Point.move(p, 3)
```

Remark 4.1.14 (**Initialization**).
We already mentioned that the class <name> may be called as a function; let's go deeply in relation to the concept of *initialization*:

- it creates an instance of the class calling implicitly the method `__init__`; the <name> call should have as many arguments as the ones of `__init__`;
- but if there is no `__init__`, *no initialization is performed i.e.* no common structure is given to *all* the instances.
  - However, we may *add attributes* to *single instances.*

```
class Time:
    def whattimeisit(self):
        return "it's " + str(self.hour) + " and " +
            str(self.min) + " minutes"
############################
>>> t = Time()
>>> print(t.whattimeisit()) # error: no hour/min in t
AttributeError: 'Time' object has no attribute 'hour'
>>> s = Time()
>>> s.hour = 10
>>> s.min = 30
>>> print(s.whattimeisit()) # no error
it's 10 and 30 minutes
```

Example 4.1.15 (**Exercise: 'class Student'**).
As an exercise:

1. define a **class** `Student` with:
   - attributes:
     - `registration number`;

  - `name`;
  - `year` (*e.g.* choose one from (1, 2, 3, 4, 5)).
- Methods:
  - `pass()`: move the student to their next year; set `year` to `None` if `year` was already 5;
  - `show()`: `print` all the attributes of the instance.

2. Create 10 instances and put them into a list.

3. Sort that list by increasing `registration number`.

Example 4.1.16 (**Expanding the 'class Point'**).
Let's consider again:

```
class Point:
    def __init__(self, xx, yy):
        self.x = xx
        self.y = yy
    def whoareyou(self):
        return self.x, self.y
    def move(self, delta):
        self.x += delta
        self.y += delta
```

Let's define a new method for adding (in a trivial way) two points; more formally:

- given two `Point` instances `p` and `q`, define a method `Psum` such that `p.Psum(q)` `return` a new `Point` instance with attributes:
  - `p.x + q.x`
  - `p.y + q.y`

So, let's define:

```
def Psum(self, other):
    return Point(self.x + other.x, self.y + other.y)
```

And considering implemented the expanded `class`:

```
>>> p = Point(1, 2)
>>> q = Point(3, 4)
>>> s = p.Psum(q)
>>> print(s.whoareyou())
(4, 6)
```

In order to saving code and multiplicity of names, it would be nice if we could overload method names with known simbols.

Remark 4.1.17 (**Special methods: overloading**).
Referring to Example 4.1.16 it would be nice if we could *overload* `+` and write `s = p + q` instead of `s = p.Psum(q)`.
So, in order to perform this kind of *overloading* we can use what are called *special methods*:

- `__add__(self, other)`: + calls the special method, if any.

  Instead of

  ```
  def Psum(self, other):
      return Point(self.x + other.x, self.y + other.y)
  ```

  Define:

  ```
  def __add__(self, other):
      return Point(self.x + other.x, self.y + other.y)
  ```

- `__str__(self, <optional args>)`: must **return** a string; is automatically called when the instance is argument of **print()**.

  In order to print an instance the following code is not much helpful:

  ```
  >>> p = Point(0, 1)
  >>> print(p)
  <__main__.Point object at 0x7fedbf6a8510>
  ```

  We could add a method:

  ```
  def Pprint(self):
      print(str(self.whoareyou()))
  ################
  >>> p = Point(0, 1)
  >>> p.Pprint()  # or Point.Pprint(p)
  (0, 1)
  ```

  But we may do much better. Define the *magic method*:

  ```
  def __str__(self):
      return 'Point ' + str(self.whoareyou())
  ################
  >>> p = Point(0, 1)
  >>> print(p)
  Point (0, 1)
  ```

- `__eq__(self, other)`: overload the == operation.

  Note that == on instances of user defined classes coincides with **is**. It cannot be otherwise! We could make the Python machine divine what the user intends to be equalty with the above special method.

  On **class** `Point` for example:

  ```
  def __eq__(self, other):
      return self.x == other.x and self.y == other.y
  ```

Definition 4.1.18 (Special methods: overloading).
Some magic methods for *comparison*:

- `__ne__(self, other)` defines !=;

- `__lt__(self, other)` defines <;
- `__gt__(self, other)` defines >;
- `__le__(self, other)` defines <=;
- `__ge__(self, other)` defines >=.

Some *numerical* magic methods:

- `__add__(self, other)` defines +;
- `__sub__(self, other)` defines −;
- `__mul__(self, other)` defines ∗;
- `__floordiv__(self, other)` defines //;
- `__truediv__(self, other)` defines /;
- `__pow__(self, other)` defines ∗∗.

More *numerical* magic methods:

- `__pos__(self)` defines the unary +;
- `__neg__(self)` defines the unary −;
- `__abs__(self)` defines function `abs()`;
- `__round__(self, n)` defines function `round()`;
- `__floor__(self)` defines function `math.floor()`;
- `__ceil__(self)` defines function `math.ceil()`;
- `__trunc__(self)` defines function `math.trunc()`.

Methods for *accessing sequences*:

- `__len__(self)` defines `len()`;
- `__getitem__(self, key)` defines `self[key]`;
- `__setitem__(self, key, value)` defines `self[key] = value`.

Methods for *augmented assignment*:

- `__iadd__(self, other)` defines the RHS of +=;
  - for instance, if `x` is an instance of a class with an `__iadd__()` method, `x += y` is equivalent to `x = x.__iadd__(y)`
- `__isub__(self, other)` defines the RHS of −=;

- *etc.*

  They should be used if you want make += *modify in place* the object at LHS, like += on `list` *etc.*

  If you want, instead, that += returns a new instance, just define `__add__(self, other)` and += will use that.[1]

NOTE: boolean operations (`and`, `or`, `not`) *cannot* be redefined through special *dunder* methods `__xxx__`.

We can also *copy* a generic object; but we can perform *different tipology* of copies. Note that assignment statements in Python do not copy objects, they create bindings between a target and an object. For collections that are mutable or contain mutable items, a copy is sometimes needed so one can change one copy without changing the other. The following module[2] provides generic *shallow* and *deep* copy operations (explained below).

Definition 4.1.19 (Copying/Cloning objects).
In order to perform a *copy* of a generic object `o1`:

1. `import copy`

   (a) `copy.copy(o1)`: returns the shallow copy of `o1`;

   (b) `copy.deepcopy(o1)`: returns the deep copy of `o1`.

The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

- a *shallow copy* constructs a new compound object and then (as far as possible) inserts references into it to the objects found in the original;
- a *deep copy* constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

NOTE: for our porpuse the difference beetween them should be not relevant, but it is important to know it.

We also mentioned the term *attribute* in relation to the concept of class; let's go deeply.

Definition 4.1.20 (Private attribute).
A *private* attribute is an attribute which:

- starts with *dunder*;
- does not end with *dunder*.

[1] These methods are called to implement the augmented arithmetic assignments (+=, −=, ∗=, *etc.*). These methods should attempt to do the operation *in-place* (modifying `self`) and `return` the result (which could be, but does not have to be, `self`). If a specific method is not defined, the augmented assignment falls back to the normal methods. For instance, if `x` is an instance of a class with an `__iadd__()` method, `x += y` is equivalent to `x = x.__iadd__(y)`. Otherwise, `x = x.__iadd__(y)` and `y.radd__(x)` are considered, as with the evaluation of `x + y`.

[2] A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended.

It is *mangled* by the Python machine, so that it appears '*private*' to the class. So, an attribute:

- `__name`

It is usable (seen) '*as is*' only in the class in which it is defined (in all its methods); it is not visible outside the class (not even in *subclasses*: we don't know yet what they are, we will see them in the next pages.)
A `__name` attribute inside the `class CLASSNAME` is *mangled* into:

- `_CLASSNAME__name`

The same definition and considerations can be done for *private* methods (see Appendix D).

Remark 4.1.21 (**Class attributes vs instance attributes**).
Consider the following code:

```
class A:
    class_attr = 10
    def __init__(self, par):
        self.a = par
    def inc(self):
        self.a += A.class_attr # Wrong if only class_attr
    def __str__(self):
        return 'A(' + str(self.a) + ')'
#####################################
>>> print(A.class_attr)
10
>>> inst = A(10)
>>> inst.inc()
>>> print(inst.a)
20
>>> print(A.class_attr)
10
```

Focus on the `class_attr` above:

- all instances of a class share an attribute; modifications to the attribute are immediately shared by all instances; this kind of attribute is called '*class attribute*' (or *static attribute*, kind of Java terminology);
- class attributes are accessed by dot notation *using the class name*: `A.class_attr`. You cannot use the non-qualified name (`class_attr`), even in the class definition.

Consider also:

```
class A:
    class_attr = 10
    def __init__(self):
        self.a = 1
#############################
>>> s = A()
>>> A.class_attr
10
```

```
>>> A.class_attr = 12
>>> p = A()
>>> p.class_attr
12
>>> p.a
1
>>> p.A.class_attr
AttributeError: 'A' object has no attribute 'A'
>>> A.class_attr
12
>>> p.class_attr = 5
>>> p.class_attr
5
>>> s.class_attr
12
>>> A.class_attr = 7
>>> p.class_attr
5
>>> s.class_attr
7
```

We conclude this paragraph saying that there is no scope nesting for class definitions.

4.2 Subclasses

How can we expand and specify the concept of *class*? It' time to introduce the concept of *subclass*; so it will be fundamental also the *inheritance* logic and, in particular, for this logic we will see the *dynamic method lookup*.

Example 4.2.1 (**Expanding a class**).
Let's consider again the `class Point`:

```
class Point:
    def __init__(self, xx, yy):
        self.x = xx
        self.y = yy
    def whoareyou(self):
        return self.x, self.y
    def move(self, delta):
        self.x += delta
        self.y += delta
```

We would like to define a *new class* with:

- same data;
- more operations:
  - a method to move the point to the origin.

```
class NewPoint:
    def __init__(self, xx, yy):
```

```
        self.x = xx
        self.y = yy
    def whoareyou(self):
        return self.x, self.y
    def move(self, delta):
        self.x += delta
        self.y += delta
    def origin(self):
        self.x, self.y = 0, 0
```

Any `NewPoint` could be seen as a `Point`; more precisely: any operation (method) on `Point` is defined (it makes sense) also on a `NewPoint`.
But for Python there is no relation between a `Point` and a `NewPoint`. Moreover, code is replicated in `Point` and `NewPoint`.
We want to make this relation explicit *i.e.* define `NewPoint` in such a way that *any instance* of `NewPoint` *is an instance* of `Point`.

Example 4.2.2 (**Defining a subclass**).
Continuing the tractation of the above Example 4.2.1, in order to reach the purpose there mentioned, we define `NewPoint` to be a `subclass` of `Point` (subclasses are defined by '*extention*'):

```
class Point:
    def __init__(self, xx, yy):
        self.x = xx
        self.y = yy
    def whoareyou(self):
        return self.x, self.y
    def move(self, delta):
        self.x += delta
        self.y += delta

class NewPoint(Point): # Note that there is no __init__
    def origin(self):
        self.x, self.y = 0, 0
```

Equivalent terminology:

- `NewPoint` is a *subclass* of `Point`;
- `Point` is the (immediate) *superclass* of `NewPoint`;
- `NewPoint` is *based* on `Point`, it is *derived* from `Point`;
- `Point` is the *base class* of `NewPoint`.

Remark 4.2.3 (**Subtyping**).
Considering implemented the code present in the Example 4.2.2:

- instances of `NewPoint` are *also* instances of `Point`;
- function `isinstance(object, class)`:
  - generalises the function `type(object)`.

```
>>> np = NewPoint()
TypeError: __init__() missing 2 required positional
arguments: 'xx' and 'yy' # see next remark
>>> np = NewPoint(4, 5)
>>> isinstance(np, Point)
True
```

Remark 4.2.4 (**Inheritance**).
Considering implemented the code present in the Example 4.2.2:

- `NewPoint` *inherits* from `Point` attributes and methods.

```
>>> np = NewPoint(1, 2) # where is __init__ for NewPoint?
>>> np.move(3) # where is move() for NewPoint?
>>> np.origin()
>>> p = Point(5, 6)
>>> p.origin() # WRONG!
AttributeError: 'Point' object has no attribute 'origin'
```

Example 4.2.5 (**Method overriding**).
If we don't like an inherited method we can *override* its definition:

```
class Point:
    def __init__(self, xx, yy):
        self.x = xx
        self.y = yy
    def move(self, delta):
        self.x, self.y = self.x + delta, self.y + delta

class NewPoint(Point):
    def origin(self):
        self.x, self.y = 0, 0
    def move(self, delta):
        self.x, self.y = self.x + 2 * delta, self.y + 2 *
            delta
```

We inherit what we don't re-define (*override*).

Example 4.2.6 (**Method 'super()'**).
A colored point:

```
class Point:
    def __init__(self, xx, yy):
        self.x = xx
        self.y = yy
    def move(self, delta):
        self.x, self.y = self.x + delta, self.y + delta

class ColPoint1(Point):
```

How can we initialize it?

```
class ColPoint1(Point):
    def __init__(self, xx, yy, col):
        self.x = xx
        self.y = yy
        self.c = col
    def origin(self):
        self.x, self.y = 0, 0
```

We don't like much this: we copied `__init__` from `Point`, plus something.
Much better: initialize *first* as a `Point`, then do more:

```
class ColPoint(Point):
    def __init__(self, xx, yy, col):
        super().__init__(xx, yy)
        self.c = col
    def origin(self):
        self.x, self.y = 0, 0
```

We used the method:

- `super()`:
  - `return` the instance where it is called *viewed* in its superclass. It is a kind of `self`, moved into its superclass.

Remark 4.2.7 (**Instances: subclasses vs superclasses**).
Remember: any instance of a *subclass* is also an instance of *any* of its *superclasses*.
Be carefull: instances of sublcasses have more information which becomes inaccessible when *viewed* from the superclass.

Definition 4.2.8 (**Method 'super()'**).
The method `super()` (with this form) viewed in the previous Example can only be used *inside* a class definition.
There is a *general* form of `super` usable everywhere:

- `super(class, obj)`;
  - `obj` must be an instance of `class`; `return` `obj` *viewed* in the *superclass* of `class`.

Remark 4.2.9 (**Dynamic method lookup**).
Consider:

```
class Point:
    ...
    def move(self, delta):
        self.x, self.y = self.x + delta, self.y + delta

class NewPoint:
    ...
    def move(self, delta):
        self.x, self.y = self.x + 2 * delta, self.y + 2 *
            delta
```

Now we call the method `move(10)` on a name:

```
who.move(10)
```

Which of the two '`move`' is actually called? It depends ... From what?
We already saw something similar:

```
S1 + S2
```

Which + is executed? Addition or concatenation or ... It depends ... From what?
When a method is called on an instance is the class of the receiving instance to determine *which* definition will be used.
The choice cannot happen on the basis of the only text of the program (*i.e. statically*).
The choice depends from the execution (it is a *dynamic* selection).
It depends from the '*type*' of **`self`**.
Methods (when called via the dot notation) behave differently from functions.

Example 4.2.10 (**Dynamic method lookup**).
Follow the dynamic logic:

```
class A:
    def __init__(self, x):
        self.aa = x
    def m1(self):
        return self.aa

class B(A):
    def m1(self):
        return super().m1() + 1
    def m2(self, x):
        return self.aa + x

class D(B):
    def m2(self, z):
        return self.aa + 2 * z
################################
>>> a = A(10)
>>> print(a.m1())
10
>>> b = B(10)
>>> print(b.m1())
11
>>> d = D(20)
>>> print(d.m1())
21
>>> print(d.m2(3))
26
```

Example 4.2.11 (**Late binding of 'self'**).
Let's consider another fundamental example, highlighting that dynamic method lookup (dispatch, selection) is the heart of object oriented programming:

```
class E:
    def __init__(self, y):
```

```
        self.ee = y
    def f(self):
        return self.ee
    def g(self):
        return self.f()

class H(E):
    def f(self):
        return 100
################################
>>> e = E(10)
>>> print(e.g())
10
>>> h = H(20)
>>> print(h.g())
100
```

Let's remove `f()` from `E`:

```
class E:
    def __init__(self, y):
        self.ee = y
    def g(self):
        return self.f()    # Correct Python!

class H(E):
    def f(self):
        return 100
##########################
>>> h = H(20)
>>> print(h.g())
100
```

We say that `E` delegates `f()` or that `g()` in `E` delegates `f()`. As consequences:

- `g` *cannot* be called on `E` instances;
- `E` must be *subclassed* before use; if not subclassed:

  ```
  >>> e = E(10)
  >>> print(e.g())
  AttributeError: 'E' object has no attribute 'f'
  ```

  Can we express inside the language this 'must be subclassed before use'? We can use `abstract classes`; let's see here just a glimpse:

  ```
  from abc import ABC, abstractmethod

  class E(ABC):
      def __init__(self, y):
          self.ee = y
      @abstractmethod      # this is a 'decorator'
      def f(self):
          pass
      def g(self):
          return self.f()
  ```

```
class H(E):
    def f(self):
        return 100
#######################################
>>> e = E(10)
TypeError: Cannot instantiate abstract class E with
abstract methods f
>>> h = H(20)
>>> print(h.g())
100
```

Definition 4.2.12 (**Abstract base classes**).
In the last part of the previous Example we saw just a glimpe of `Abstract base classes`. They are classes where:

- some methods are defined without (complete) implementation;
- complete body for the *abstract method* should be provided in subclasses.

`Abstract base classes` cannot be instantiated. They are `interfaces`: names of operations vs implementation of those names.

Example 4.2.13 (**Abstract base classes: blueprints, interfaces**).
Abstract base classes are effectively `interfaces`, `blueprints`:

```
from abc import ABC, abstractmethod

def MyAbstract(ABC):
    @abstractmethod
    def method1(self):
        pass
    @abstractmethod
    def method2(self):
        pass

def MyConcrete(MyAbstract):
    def method1(self):
        real implementation
    def method2(self):
        real implementation
```

A real contract `MyConcrete` says: 'I will respect the `interface` `MyAbstract`'.

Remark 4.2.14 (**Abstract base classes are interfaces**).
ABCs are `interfaces`: names of operations vs implementation of those names.
In Java:

- `interfaces` are an indipendent concept:
  - lists of names (and types) of methods;
- A Java class may:
  - extends a (super) class;

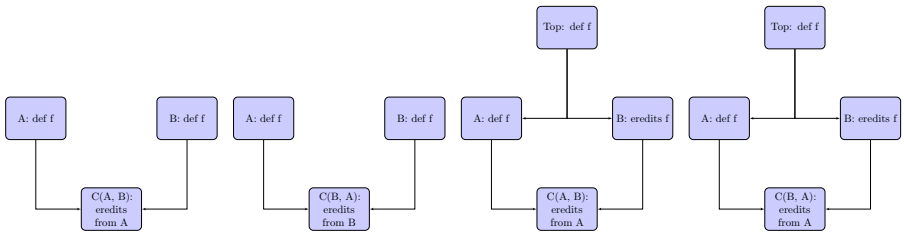


Figure 4.1: multiple inheritance logics.

– implements an interface: provides implementations for the methods.

So far we have seen the generic concept of Python inheritance. Let's go deeply.

Remark 4.2.15 (**Single inheritance**).
In a single inheritance logic any class is a subclass of a *single immediate* superclass. That is, in any class definition:

```
class A(superclass):
    body
```

There is *at most one* `superclass` listed in the header.

Remark 4.2.16 (**Multiple inheritance**).
A class definition may list *more than one* superclass:

```
class A(B, C):
    body
```

Hence `A` is subclass of both `B` and `C` (which in general are not related by a super/subclass relation between them).
Following multiple inheritance logic a class inherits from more than one *immediate* superclass.
Conceptually simple, but raises problems with non trivial solutions:

- who is `super()` if we have more than one superclass?
- Which method is inherited if defined in more than one immediate superclass?
- *Etc.*

Look in the documentation for *MRO*: *Method Resolution Order*. This is the official method resolution order defined by the language.
Class method: `class.mro()`.

Let's see some logics about Python method resolution order.

Example 4.2.17 (**Multiple inheritance**).
Multiple inheritance generally depends on the order of superclasses seen as '*parameters*' of a subclass (for a graphical view see Figure 4.1):

```
class A:
    def f(self):
        return 7

class B:
    def f(self):
        return 8

class C(A, B):
    pass
################################
>>> c = C()
>>> c.f()
7
```

And

```
class C(B, A):
    pass
################################
>>> c = C()
>>> c.f()
8
```

But:

```
class Top:
    def f(self):
        return 1

class A(Top):
    def f(self):
        return 7

class B(Top):
    pass

class C(A, B): # obviously C is also a subclass of Top
    pass       # inheritance is transitive
################################
>>> c = C()
>>> c.f()
7
```

And:

```
class C(B, A):
    pass
################################
>>> c = C()
>>> c.f()
7
```

Remark 4.2.18 (MRO is not always defined).
There are cases where the MRO is not defined. Hence they are not legal programs. Example:

```
class CC:
    pass
class DD:
    pass
class AA(CC, DD):
    pass
class BB(DD, CC): # consistent if we swap DD, CC
    pass
class EE(AA, BB): # if we cut this gives no error
    pass
# after running the message error is
class EE(AA, BB):
TypeError: Cannot create a consistent method resolution
order (MRO) for bases CC, DD
```

Chapter 5

Errors handling

5.1 Exceptions

As for other programming languages, also for Python is possible managing *errors*.

Example 5.1.1 (Errors/Exceptions: raising).
Errors/Exceptions come in various '*kinds*':

```
>>> print(1/0)
ZeroDivisionError: division by zero
##################################
>>> L = [1, 2, 3]
>>> print(L[10])
IndexError: list index out of range
```

The correct terminology is '*raising an exception/error*'.

But talking about *errors* is the same as talking about *exceptions*?

Remark 5.1.2 (Why exception and not error?).
Talking about *errors* is too general; we may have different type of *error*:

- *syntactic*:

  ```
  if 0 == 1:
      print('NO!') # never go inside
  ```

- *semantic*:

  ```
  while True:
      pass # always go inside
  ```

- *dynamic*:

  ```
  >>> print(1/0) # this raises an exception
  ZeroDivisionError: division by zero
  # the ZeroDivisionError exception
  ```

So we can say that *exceptions* are a subset of *errors*, and they are *tangible* by Python which makes them explicitly visible to the programmer.

But what does concretely specialize *exceptions* if compared to generic *errors*?

Remark 5.1.3 (**Handle an exception**).
We may *handle* an *exception* with a **try/except** command:

```
try:
    print(1/0)
except ZeroDivisionError:
    print('OK!')
print('after')
#########################
OK!
after
```

The `ZeroDivisionError` *exception* is *caught* by the **except** clause.
But be carefull:

```
try:
    print(1/0)
except IndexError:
    print('OK!')
print('after')
#########################
ZeroDivisionError: division by zero
```

The `ZeroDivisionError` *exception* makes the program terminate, because is *not caught* by the **except**.

Definition 5.1.4 (**'try/except' command**).
Let's define the **try/except** command:

```
try:
    <block>
except <exception>:
    <handlerblock>
######
<after>
```

1. `<block>` is executed:
   (a) if *no* `<exception>` occurs (and there aren't other errors), then execution proceeds to `<after>`;
   (b) if `<exception>` occurs, `<handlerblock>` is executed; then (if there aren't other errors) execution proceeds to `<after>`.

If the **except** part of the command is simply:

```
except:
    <handlerblock>
```

This catches *all* exceptions (so it catches the first one found), while the portion of `<block>` evaluated before the exception mantains its effect:

```
>>> try:
      X = 10
      Y = 100 / 0
      X = 100
    except:
      Y = 200
>>> X
10
>>> Y
200
```

Example 5.1.5 (**A programming example**).
We input integers and print their square, until 'stop' is read. If a non integer is read, signal the error and proceed with next input.
Without **try/except** command:

```
reply = input('Enter text (or "stop"): ')
while reply != 'stop':
    if not reply.isdigit():
        print('Bad!' * 8)
    else:
        print(int(reply) ** 2)
    reply = input('Enter text (or "stop"): ')
print('Bye')
####################
Enter text (or "stop"): 7
49
Enter text (or "stop"): s
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
Enter text (or "stop"): stop
Bye
```

With **try/except** command:

```
reply = input('Enter text (or "stop"): ')
while reply != 'stop':
    try:
        print(int(reply) ** 2)
    except:
        print('Bad!' * 8)
    reply = input('Enter text (or "stop"): ')
print('Bye')
```

Or another equivalent portion of code which highlights that Python does not have a <`do ... until`> command, but the correspondent idiomatic coding in Python would be <**`while`** `True ...` **`break`**>:

```
while True:
    reply = input('Enter text (or "stop"): ')
    if reply == 'stop':
        break
    try:
        print(int(reply) ** 2)
    except:
```

```
        print('Bad!' * 8)
print('Bye')
```

Definition 5.1.6 ('try/except' general command).
The **try/except** general command is:

```
try:
    <block> # run this first
except name_1:
    <block_1> # run if name_1 is raised in try block
except (name_2, name_3):
    <block_2> # run if any of these excs raised in try
except name_4 as name:
    <block_4> # if name_4 raised, assign instance to name
except:
    <block_5> # run for all other exceptions raised
else:
    <block_6> # run if no excs raised in try block
########
<after>
```

The commands are read in the following way:

1. <block> is executed:

   (a) the first exception found (is *raised*) – if any – in the set of (name_1, name_2, name_3, name_4, other exceptions, no exception) determines which block will be executed (the other are not *raised*).

   Note: name_1 is not *raised* if the first exception that we found in the **try** is name_2, even if name_1 is also present in the **try**, but follows name_2.

Example 5.1.7 ('try/except' general command).
Some examples to understand the logic above defined; consider the twos:

```
L = [1, 2, 3]
>>> L[10] = 5
IndexError: list assignment index out of range
>>> 1/0
ZeroDivisionError: division by zero
```

And

```
try:
    x = 10
    y = 1 / 0
    L = [1, 2, 3]
    L[10] = 5
except IndexError:
    print('index out')
except ZeroDivisionError:
    print('zero div')
else:
    print('no exceptions')
print('after')
```

```
##########################
zero div
after
>>> x
10
>>> L
NameError: name 'L' is not defined
```

And:

```
try:
    L = [1, 2, 3]
    L[10] = 5
    x = 10
    y = 1 / 0
except IndexError:
    print('index out')
except ZeroDivisionError:
    print('zero div')
else:
    print('no exceptions')
print('after')
#########################
index out
after
>>> L
[1, 2, 3]
>>> x
NameError: name 'x' is not defined
```

And:

```
try:
    L = [1, 2, 3]
    L[10] = 5
    x = 10
    y = 1 / 0
    z = 7
except (ZeroDivisionError, IndexError):
    print('index out or zero div')
else:
    print('no exceptions')
print('after')
###########################
index out or zero div
after
>>> L
[1, 2, 3]
>>> x
NameError: name 'x' is not defined
>>> z
NameError: name 'z' is not defined
```

Remark 5.1.8 (**Exception values**).
In the Definition 5.1.6 we wrote this portion of code:

```
try:
    <block> # run this first
...
except name_4 as name:
    <block_4> # if name_4 raised, assign instance to name
```

Exceptions usually come also with a *value*:

```
>>> 1 / 0
ZeroDivisionError: division by zero
```

For example the standard `ZeroDivisionError` comes with the *string value* `division by zero`

```
try:
    print(1 / 0)
except ZeroDivisionError as ExceptionValue:
    print(ExceptionValue)
    print('OK')
######################
division by zero
OK
```

Or others:

```
>>> '2' + 2
TypeError: can only concatenate str (not "int") to str
#####################
>>> 2 + '2'
TypeError: unsupported operand type(s) for +: 'int' and
'str'
```

Remark 5.1.9 (**The 'else' clause**).
Considering:

```
try:
    <block> # run this first
except name_1:
    <block_1> # run if name_1 is raised in try block
except (name_2, name_3):
    <block_2> # run if any of these excs raised in try
except name_4 as name:
    <block_4> # if name_4 raised, assign instance to name
except:
    <block_5> # run for all other exceptions raised
else:
    <else_block> # run if no excs raised in try block
########
<after>
```

Why having an **else** `clause`? We could include the `<else_block>` into the `<after>`.
Afterall `<after>` could be also run if *no exception is raised* in **try** `block` (which is the condition to run the `<else_block>`; better explained: if the condition

to run the <else_block> is satisfied and the <else_block> has no error $\Longrightarrow$ then also <after> is reached; but it can be that <after> is reached without that the <else_block> is executed).
Indeed the code:

```
try:
    <block>
except:
    ...
else:
    <else_block>
#####
<after>
```

Is very much like:

```
try:
    <block>
    <else_block>
except:
    ...
#####
<after>
```

But what if <else_block> *raises an exception*? Did we get at <after> because the try failed or not?

```
try:
    x = 5
except:
    print('zero')
else:
    y = 1 / 0
print('after')
######################
ZeroDivisionError: division by zero
```

In the above code we didn't reach the `print('after')` command while in the following succeedes:

```
try:
    x = 5
    y = 1 / 0
except:
    print('zero')
print('after')
#####################
zero
after
```

Definition 5.1.10 (**The 'finally' clause**).
In order to complete Definition 5.1.6 consider also the following structure which introduces a new clause:

```
try:
```

```
    <block> # run this first
except name_1:
    <block_1>
...
else:
    <else_block> # run if no exception was raised in try
finally:
    <final_block> # run always before leaving the try
```

- `<final_block>` is executed in any case, as final code, before leaving the **`try`** either if exceptions are raised or not, are handled or not.

Example 5.1.11 (The 'finally' clause).
In order to fully understand the **`finally`** clause defined in the previous Definition, consider the following:

```
try:
    print(1 / 0)
finally:
    print('I am here in any case')
##################################
I am here in any case
ZeroDivisionError: division by zero
```

And

```
try:
    L[10]
    print(1 / 0)
except ZeroDivisionError as ExceptionValue:
    print(ExceptionValue)
print('OK')
##########################################
NameError: name 'L' is not defined
```

In comparison to:

```
try:
    L[10]
    print(1 / 0)
except ZeroDivisionError as ExceptionValue:
    print(ExceptionValue)
finally:
    print('OK')
##########################################
OK
NameError: name 'L' is not defined
```

We use **`finally`** to release resources: disconnect from network, close/release files or locks, *etc.*

Remark 5.1.12 (Exceptions and the frame stack).
A normal programming situation: the exception *will not appen directly in the* **`try`** `block` but *in a function/method called from inside of the* **`try`** `block`.

```
try:
    f()
except Excpname:
    print('Excpname caught')
```

Furthermore it could happen that `f()` may call `g()` which may call `h()` and only at that point `Excpname` is raised:

```
def f():
    a = 0
    g()
    return 1
def g():
    h()
    return 2
def h():
    c = 0
    return 1 / 0 # ZeroDivisionError raised

try:
    print(f())
except ZeroDivisionError:
    print('ZeroDivisionError caught')
#########################################
ZeroDivisionError caught
```

Exceptions are propagated through the stack.
Frames are popped out of the stack if their function does not handle the exception.
The code remaining in those function is not executed.
If an exception is never caught, the exception is passed to the standard exception handler of the `__main__`:[1] the program terminates and the exception is printed.

Remark 5.1.13 (**Raising exceptions explicity**).
We may *explicitly raise* an exception.
The command:

- `raise` <`exception_instance`> raises the exception.

```
#### prova.py
print('first')
raise IndexError
print('second')
#########################
>>> %Run prova.py
    first
    IndexError
```

[1] `__main__` is the name of the environment where top-level code is run. 'Top-level code' is the first user-specified Python module that starts running. It's 'top-level' because it imports all other modules that the program needs. Sometimes 'top-level code' is called an entry point to the application. The top-level code environment can be the scope of an interactive prompt: >>>`__name__` prints '`__main__`'.

Remark 5.1.14 (Built-in exceptions).
Some built-in exceptions:

- `ZeroDivisionError`;
- `IndexError`;
- `KeyError`;
- `TypeError`;
- `ValueError`;
- `NameError`;
- `OSError`.

Definition 5.1.15 (User defined exceptions).
An exception is an instance of (a subclass of) `Exception`:

```
class MyExcp(Exception):
    pass    # enough just define
...
try:
    # other code
    raise MyExcp # short for MyExcp()
except MyExcp:
    print('caught')
```

We may add a *value* to the instance, so that we may use that value in the handler:

```
class MyExcp(Exception):
    pass

try:
    raise MyExcp(10)
except MyExcp as X:
    print('caught', X)
###################### run
caught 10
```

In an `except ... as <name>` clause, `<name>` is local to the handler and is destroyed when the handler terminates:

```
class MyExcp(Exception):
    pass
...
try:
    raise MyExcp(10)
except MyExcp as X:
    print('caught', X) # handler
######################
print(X) # fails
```

The argument of `raise` is *any reference* to an *exception object*:

```
class MyExcp(Exception):
    pass
P = (MyExcp, IndexError)

try:
    raise P[0]
except MyExcp:
    print('caught')
```

The argument of **except** is *any exception class*:

```
class MyExcp(Exception):
    pass

try:
    raise MyExcp
except Exception: # same as except:
    print('caught')
```

Remark 5.1.16 (**Subclasses of user defined exceptions**).
A clause:

- **except** <exceptionclass>
  - will catch *all exceptions* which are instances of <exceptionclass> (therefore all instances of its subclasses).

```
class MyExcp(Exception): pass
class MySpec1(MyExcp): pass
class MySpec2(MyExcp): pass

for e in (MyExcp(0), MySpec1(1), MySpec2(2)):
    try:
        raise e
    except MyExcp as X:
        print('caught', X)
##########################################
caught 0
caught 1
caught 2
```

So, what about hierarchies?

Remark 5.1.17 (**Hierarchies of user defined exceptions**).
In order to understand the hierarchies of user defined exceptions consider the followings:

```
class B(Exception): pass
class C(B): pass
class D(C): pass

for cls in [B, C, D]:
    try:
```

```
        raise cls
    except D:
        print('D')
    except C:
        print('C')
    except B:
        print('B')
###################
B
C
D
```

But:

```
class B(Exception): pass
class C(B): pass
class D(C): pass

for cls in [B, C, D]:
    try:
        raise cls
    except B:
        print('B')
    except D:
        print('D')
    except C:
        print('C')
###################
B
B
B
```

Essentially the reasoning is: `B`, `C` and `D` are all instances of `B` (at the top level), so the order in the sequence of **`except`** becomes determining.

Example 5.1.18 (Programming with exception: part 1).
Compute the product[2] of a sequence of integers:

```
# Iteration 1
def prod(S):
    res = 1
    for e in S:
        res *= e
    return res
```

What about stopping when a zero is found?

```
# Iteration 2
def prod(S):
    res = 1
    for e in S:
        if e == 0:
            return 0
        res *= e
    return res
```

[2]The `math` module has `math.prod`, only from Python 3.8.

return will break the iteration and terminate the function.
Now compute again the product of a sequence of integers using recursion, instead of iteration:

```
# Recursion 1
def recprod(S):
    if len(S) == 0:
        return 1
    else:
        return S[0] * recprod(S[1 :])
```

What about stopping when a zero is found?

```
# Recursion 2
def recprod(S):
    if len(S) == 0:
        return 1
    else:
        if S[0] == 0:
            return 0
        return S[0] * recprod(S[1 :])
```

It is correct but not quite the same as the *iterative* **return 0**; it will stop further recursive calls, but it will be multiplied with the *previous* elements of S.
We must propagate that zero along the chain of the recursive calls:

```
# Recursion 3
class Zero(Exception): pass

def recprod(S):
    def aux(S):
        if len(S) == 0:
            return 1
        else:
            if S[0] == 0:
                raise Zero
            else:
                return S[0] * aux(S[1 :])
    try:
        res = aux(S)
    except Zero:
        return 0
    else:
        return res
```

Example 5.1.19 (**Programming with exception: part 2**).
Breaking out multiple nested loops:

```
class Exitloop(Exception): pass

try:
    while True:
        while True:
            for i in range(10):
                if i > 3:
```

```
                    raise Exitloop   # break doesn't work
                print('loop3: ', i)  # ther're 2 while
            print('loop2')
        print('loop1')
except Exitloop:
    print('caught')
################################
loop3:  0
loop3:  1
loop3:  2
loop3:  3
caught
```

Chapter 6

Iterators and computational complexity

6.1 Iterators in-depth

Remark 6.1.1 (Generators recall).
We have seen (see from Definition 3.3.60) *generators*; as generator expressions:

- `G = (2 * i for i in range(10))`;
- or through functions with the `yield` command.

They are implicit ways to define sequences. Elements are '*generated*' by calling `next()` on the generator.
An important point (to remember): we may use generators as sequences in `for` statements:

```
for g in G: # G is a generator
    print(g)
```

Generators are special cases of *iterators*

Definition 6.1.2 (Iterator).
An *iterator* is the most general form of sequence that may appear in a `for` command/statement:

```
for <name> in <iterator>:
    <body>
```

At any iteration, a next value is extracted from the iterator, until it is exhausted.
An *iterator* is *any instance of a class defining the two methods*:

- `__iter__`:
  - should `return` the *object itself*;
- `__next__`:
  - should `return` the *next value* of the iterator.

Example 6.1.3 (**An iterator**).
An example of a generic iterator:

```
class Even: # we iterate on even int in an interval
    def __init__(self, start, stop):
        if start % 2 != 0: # ensure start is even
            start += 1
        self.x = start
        self.stop = stop
    def __iter__(self): # return the iterator
        return self
    def __next__(self): # return the 'next' value
        if self.x < self.stop:
            curr = self.x
            self.x += 2
            return curr
        else:
            raise StopIteration
##################################################
F = Even(1, 6)

for i in F:
    print(i) # 2, 4

for i in F: # F is already exhausted
    print('here I am') # never executed

# or, maybe better we could have written
for i in Even(1, 6):
    print(i)
##################################################
>>> G = Even(1, 6)
>>> G.start # Note: above start has no self
AttributeError: 'Even' object has no attribute 'start'
>>> G.x
2
>>> G.stop
6
>>> G.curr # Note: above curr has no self, we can't
           # access it by dot
AttributeError: 'Even' object has no attribute 'curr'
```

Remark 6.1.4 (**Iterator's dynamic**).
Consider again implemented the **class** `Even` from the previous Example:

```
F = Even(1, 6)

for i in F:
    print(i) # 2, 4
```

Let's analyze what happens:

1. when the heading of **for** is evaluated, it calls `__iter__` on `F` and store away ('*freezes*') this object;

2. at any iteration, `__next__` is called on that object;
3. iteration stops when the exception `StopIteration` is raised.

Remark 6.1.5 (**Iterators out of standard sequences**).
Consider now iterators out of standard sequences (iterables):

```
for i in (1, 2, 3):
    print(i)
```

Let's analyze what happens:

1. when the heading of **for** is evaluated, it calls `__iter__` on the sequence and store away ('*freezes*') this object:
   - `(1, 2, 3).__iter__()`;
2. at any iteration, `__next__` is called on that object;
3. iteration stops when the exception `StopIteration` is raised.

Remark 6.1.6 (**Generators are iterators**).
The language handles *generators* in such a way to define for them the methods:

- `__iter__`
- `__next__`

For `G` generator, `next(G)` is nothing but `G.__next__()`.
The fact that generators are iterators may be expressed inside the language through the class hierarchy (we see this in a moment).

So, when programming, it comes natural the question:

- when using:
  - simple sequences;
  - generators;
  - more general iterators.

Example 6.1.7 (**Programming: generators vs iterators**).
Consider again the iterator:

```
class Even: # we iterate on even int in an interval
    def __init__(self, start, stop):
        if start % 2 != 0: # ensure start is even
            start += 1
        self.x = start
        self.stop = stop
    def __iter__(self): # return the iterator
        return self
    def __next__(self): # return the 'next' value
        if self.x < self.stop:
            curr = self.x
            self.x += 2
            return curr
        else:
            raise StopIteration
```

An equivalent generator could be:

```
def EvenG(start, stop):
    if start %2 != 0
        start += 1
    return (i for i in range(start, stop, 2))

for k in EvenG(1, 6):
    print(k)
```

Or just forget about generators/iterators and use simple programming.

Remark 6.1.8 (**Programming: generators vs iterators**).
Our toy case:

- generator is clearer, simpler in comparison to a generic iterator;
- even plain sequences would work.

So, in general:

- use `generators` (vs `simple sequences`) when the *generated* sequence is complex: *e.g.* the prime numbers, Fibonacci numbers, those strings which starts with a letter and ... *etc.*
- Use `iterators` (vs `generators`) when you need other methods besides `__iter__` **and** `__next__` in their class.

Example 6.1.9 (**A more generic iterator**).
The following is also an iterator:

```
class MySomething:
    def __init__(self, start, stop):
        ...
    def __iter__(self): # return the iterator
        ...
    def __next__(self): # return the 'next'
        ...
    def myothermethod_1(self, x1, x2):
        ...
    def myothermethod_2(self):
        ...
```

Remark 6.1.10 (**Abstract Base Classes and iterators**).
How to express inside the language that a class defines an iterator? That is, that it defines `__iter__` and `__next__` methods?
Remember: Abstract Base Classes (see from Definition 4.2.12):

- some methods are *defined without implementation*;
- *implementation* is provided *in subclasses*;
- Abstract Base Classes *cannot be instantiated*;
- they are a *kind of interfaces*.

The ABC for iteration is a defined library.

Example 6.1.11 (**Abstract Base Classes and iterators**).
The following:

```
class Even: # we iterate on even int in an interval
    def __init__(self, start, stop):
        if start % 2 != 0: # ensure start is even
            start += 1
        self.x = start
        self.stop = stop
    def __iter__(self): # return the iterator
        return self
    def __next__(self): # return the 'next' value
        if self.x < self.stop:
            curr = self.x
            self.x += 2
            return curr
        else:
            raise StopIteration
```

Becomes more precisely:

```
import collections.abc

class Even(collections.abc.Iterator)
# class Even:
    def __init__(self, start, stop):
        if start % 2 != 0: # ensure start is even
            start += 1
        self.x = start
        self.stop = stop
    def __iter__(self): # return the iterator
        return self
    def __next__(self): # return the 'next' value
        if self.x < self.stop:
            curr = self.x
            self.x += 2
            return curr
        else:
            raise StopIteration
##########################################
>>> Even(3, 6)
<__main__.Even object at 0x7fbfae6a8c50>
```

But:

```
import collections.abc

class Even(collections.abc.Iterator)
# class Even:
    def __init__(self, start, stop):
        if start % 2 != 0: # ensure start is even
            start += 1
        self.x = start
        self.stop = stop
```

```
    def __iter__(self):
        return self
##########################################
Even(3, 6)
TypeError: Cannot instantiate abstract class Even with
abstract methods __next__
```

And:

```
import collections.abc

class Even(collections.abc.Iterator)
# class Even:
    def __init__(self, start, stop):
        if start % 2 != 0: # ensure start is even
            start += 1
        self.x = start
        self.stop = stop
    def __next__(self):
        if self.x < self.stop:
            curr = self.x
            self.x += 2
            return curr
        else:
            raise StopIteration
##########################################
>>> Even(3,6)
<__main__.Even object at 0x7fbfae6a8c50>
>>> Even(3,6).__next__()
4
# But note without __iter__
>>> for i in Even(3, 6):
      print (i)

TypeError: 'Even' object is not iterable
```

6.2 Computational complexity of operations

Everithing in life has a cost and *time* is maybe the most expensive money; computer science in not an exception of this universal law.

Definition 6.2.1 (**Time complexity of a program**).
Time complexity of a program/function:

- time needed to run it on data `D` *as a function* of the *size* of `D`;
- instead of *actual time*, count the *number of elementary steps*;
- if elementary steps are all *constant-time* operations, the estimate is *proportional* to the actual time.

Example 6.2.2 (**Time complexity**).
Given:

```
max = 0
for e in S: # single iteration handling
    if e > max: # guard evaluation
        max = e # if assignment
```

Let's consider some aspects:

- dimension of `S`: number `n = len(S)` of elements;
- `if` assignment, single iteration handling, guard evaluation are all constant-time operations; global evaluation is proportional to n (*linear time*).

Example 6.2.3 (**Time complexity**).
Given:

```
def maxi(S):
    max = 0
    for e in S: # single iteration handling
        if e > max: # guard evaluation
            max = e # if assignment
    return max

def maxind(S):
    res = []
    for i in range(len(S)): # single iteration handling
        if S[i] == maxi(S): # guard evaluation (n x n)
            res += [i] # if assignment
    return res
```

Let's consider some aspects:

- dimension of `S`: number `n = len(S)` of elements;
- `if` assignment, single iteration handling, guard evaluation are all constant-time operations; global evaluation is proportional to n^2 (*quadratic time*).

Example 6.2.4 (**Time complexity: much better**).
Given (much better in comparison to the previous Example):

```
def maxi(S):
    max = 0
    for e in S:
        if e > max:
            max = e
    return max

def maxind(S):
    m = maxi(S) # note the position of maxi(S)
    res = []
    for i in range(len(S)):
        if S[i] == m: # (n + n = 2n)
            res += [i]
    return res
```

Let's consider some aspects:

- dimension of `S`: number `n = len(S)` of elements;
- `if` assignment, single iteration handling, guard evaluation are all constant-time operations; global evaluation is proportional to n (*linear time*).

So how does *an operation* cost? Considering:

- `if` assignment;
- single iteration handling;
- guard evaluation.

We saw are all constant-time operations.
Now considering:

- arithmetical operations;
- access to data structures;
- ...

Are all constant-time operations ... Are they in Python? Why Python would look suspicious? Let's also consider some logics of other languages and then return to Python.

Remark 6.2.5 (**Integers in other languages**).
Integers in C, Java, Pascal, Fortran, *etc.* are characterized by *one-to-one correspondence*:

- data type of the language (integer);
- internal representation on the fixed-size words of machine;
- *minimum* and *maximum* representable integer.

Remark 6.2.6 (**Arrays in other languages**).
Arrays in C, Java, Pascal, Fortran, *etc.*:

- simple sequences of values of basic types (int/float);
- straightforward mapping between array indexes and memory addresses.

Now, suppose that a *machine word* is of 4 bytes (32 bits); so for a 1-dimensional array `A` of `i + 1` integers, indexed from 0 to `i` and stored at address `b` we have that:

- `A[0]` is at address `b`;
- `A[1]` is at address `b + 4`;
- `A[2]` is at address `b + 8`;
- ...
- `A[i]` is at address `b + 4 * i`.

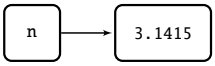
So *from index to address* is a simple computation via `b` and `4` (the *stride*): `b + 4 * i`, this formula is *known statically.*
What about multi-dimensional arrays?
For a multi-dimensional array `B` ($3 \times 3 \times 2$, `B[i, j, k]`) the formula is slightly more complicated:

- `B` is stored *by rows* (C, Java) or *by columns* (Fortran);
- generalized notion of *stride*: $(24, 8, 4)$ (*by rows*).

The formula is *known statically* `b + i * 24 + j * 8 + k * 4` via `b` and the *stride* $(24, 8, 4)$ (*by rows*).

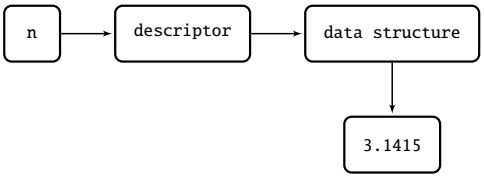
Remark 6.2.7 (**Python vs other languages**).
In summary in C, Java, Pascal, Fortran, *etc.* (*e.g.* `n = 3.1415` see figure below):

- straightforward mapping between language values and machine addresses.



In Python (*e.g.* `n = 3.1415` see figure below):

- *access* is always *indirect*;
- *name* referes to a *descriptor* which refers to a *data structure* which contains the *value(s).*



In Python *flexibility* is paid by *space* and *indirection.* Why Python is different:

- integers of *arbitrary size*;
- *variable lenght* data structure (**`list`**, **`dict`**).

While the lower-level machine has:

- *fixed size* memory words;
- *fixed correspondence* between *addresses* and *memory words.*

Main distinction:

- *constant-time* operations;
- *variable-time* operations: time depends on the *size* of the arguments.

How Python data is represented on a lower-level fixed-memory word machine?

- One Python **`int`** does not fit in a single memory word;
- **`del`** `L[i]` involves the '*recomputations of the indexes*'.

In general, for any programming language we have different interaction levels that involve the languages themselves; so we mainly have to distinguish the implementation of the language engine from the programming language itself.

Remark 6.2.8 (**Language implementation reference**).
CPython is the most common Python implementation;[1] see some fundamental aspects:

- written in C;
- simple parser[2] (of LL[3] type);
- compiler to Python bytecode;
- interpreter of Python bytecode;
- run-time support (*e.g* garbage collector[4]).

Let's inspect briefly what happens in a (Python ambient) lower level.

Remark 6.2.9 (**The underlying machine**).
We can think of the (Python) *underlying* machine as:

- organized in fixed size words:
  - in bynary: 32 bits;
  - hence 2^{32} different possible numbers.
- We access any word in constant time through its (memory) address.

For simplicity of exposition:

- examples in decimal;
- assuming words of 8 decimal digits.

---

[1] *Jython*, *IronPython* and *PyPy* are the current other implementations of the Python programming language; these are implemented in Java, C-Sharp and RPython (a subset of Python), respectively.

[2] In computer technology, a parser is a program that's usually part of a compiler. It receives input in the form of sequential source program instructions, interactive online commands, markup tags or some other defined interface. Parsers break the input they get into parts such as the nouns (objects), verbs (methods), and their attributes or options. These are then managed by other programming, such as other components in a compiler. A parser may also check to ensure that all the necessary input has been provided. A parser is a program that is part of the compiler, and parsing is part of the compiling process. Parsing happens during the analysis stage of compilation.

[3] In computer science, an LL parser (Left-to-right, leftmost derivation) is a top-down parser for a restricted context-free language. It parses the input from Left to right, performing Leftmost derivation of the sentence.

[4] In computer science, garbage collection (GC) is a form of automatic memory management. The garbage collector attempts to reclaim memory which was allocated by the program, but is no longer referenced; such memory is called garbage. Garbage collection was invented by American computer scientist John McCarthy around 1959 to simplify manual memory management in Lisp. Garbage collection relieves the programmer from doing manual memory management, where the programmer specifies what objects to de-allocate and return to the memory system and when to do so.

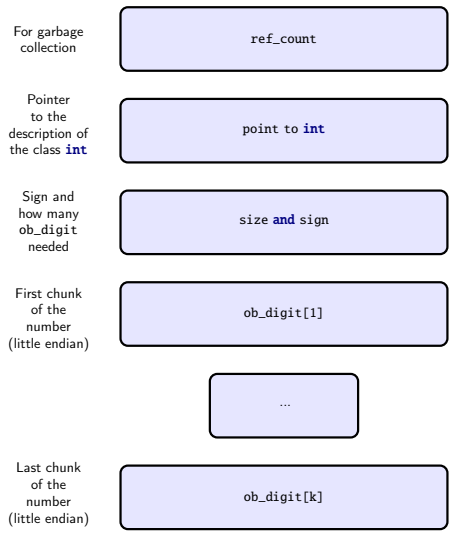


Figure 6.1: low level Python integer.

Example 6.2.10 (Integers low representation).
How are `int` values represented?
They cannot be mapped directly into the '*integers*' of the underlying machine!
We need to think them as (see Figure 6.1):

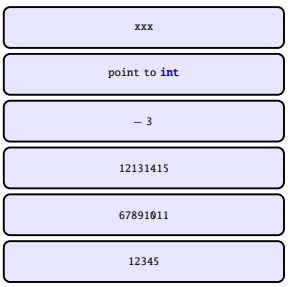
- objects;
- garbage collected;
- arbitrary size.

In order to understand better let's encode a single Python `int` *e.g.*

– `12345678910111213141 5`

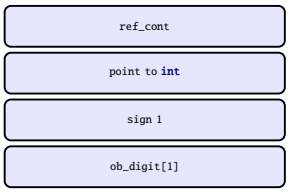
- divide it in chunks of 8 decimal digits (in realty: 30 bits) starting from right (little endian[5]):
  - – `12345 67891011 12131415`;
  - 3 chunks, negative sign:
    1. `12131415`
    2. `67891011`
    3. `12345`

[5] Big-endian is an order in which the 'big end' (most significant value in the sequence) is stored first, at the lowest storage address. Little-endian is an order in which the 'little end' (least significant value in the sequence) is stored first.



Remark 6.2.11 (**Memory allocation and optimization**).
Any time an **`int`** is used, an object of that shape is allocated in the memory using also optimization logics:

- optimization 1:
  - integers in the range $[-2^{30}, 2^{30}]$ are allocated as in the image below;
  - underlying arithmetic on `ob_digit[1]` is used if possible.



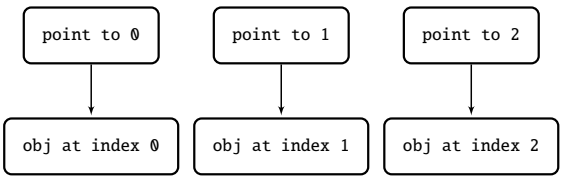
- Optimization 2:
  - small integers in the range $[-5, 256]$ are pre-allocated at initialization time;
  - any such object is never duplicated;
  - we may check this in Python through **`id`**`()`.

Example 6.2.12 (**'list' low representation**).
Let's consider:

- `[1, 2, 3]`;
- `[(1, 2, 3), 2, 'bologna']`.

They are both **`list`** of length 3. They have a common representation, characterized by:

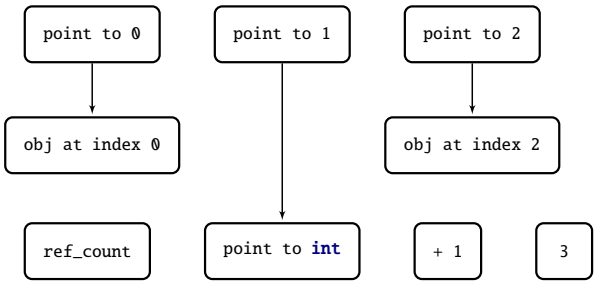
- *indirection*;
- a structure[6] of 3 links/pointers to the object members.



[6]With diagrams of below shape transpose the top-down logic of Figure 6.1 to a left-right one.

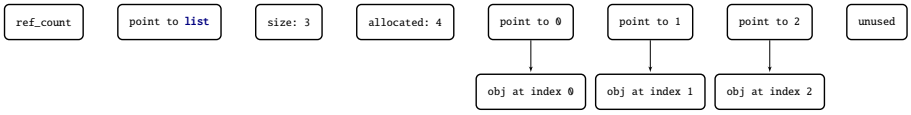
Example 6.2.13 (**'list' low representation**).
With the logic of the previous Example, let's consider:

- `[ob_0, 3, ob_2]`.



Example 6.2.14 (**'list' low representation**).
With the logic of the previous Examples, let's consider a three element **`list`** with a more complete picture:

- `[ob_0, ob_1, ob_2]`



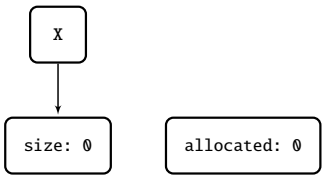
Example 6.2.15 (**Low representation: empty 'list'**).
With the logic of the previous Examples, let's consider the empty **`list`** (from now on we forget `ref_count` and `point to` **`class`**):

- `X = []`

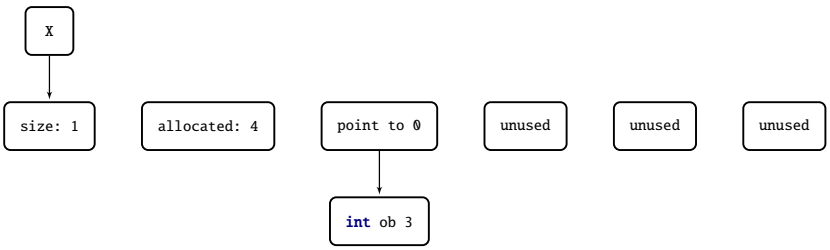
Constant-time op.

Example 6.2.16 (**Low representation: appending on empty 'list'**).
With the logic of the previous Examples, let's consider the method `append()` on an empty **`list`**:

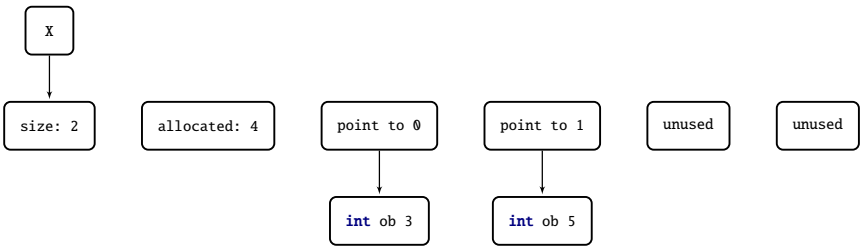
- `X = []`



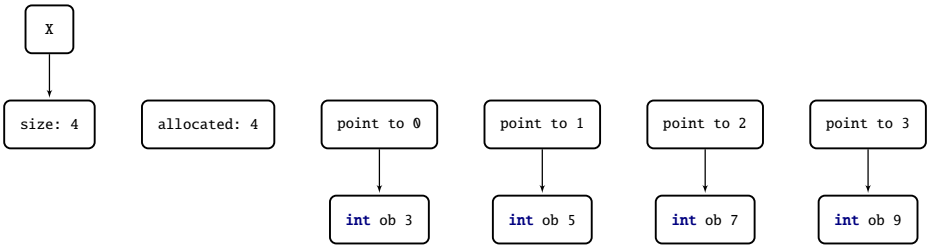
- `X.append(3)`



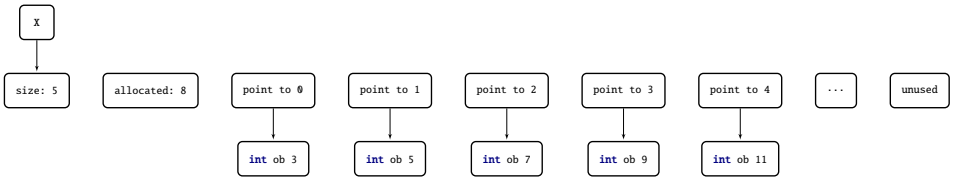
- `X = [3];`
- `X.append(5)`



- `X = [3, 5, 7, 9]`



- `X.append(11)`



Remark 6.2.17 (**Python 'list': low dynamic of 'append()' method**). Considering the Examples above we give here some description of the `append()` method low dynamic:

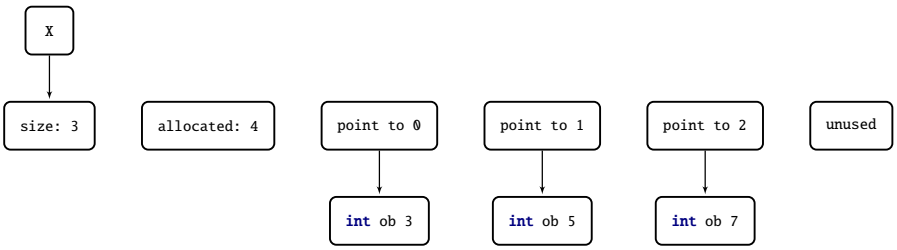
- if there is an *unused* slot for the new object (pointer): use it;
- otherwise, over-allocate a chunk of pointers; use the first:
  - over-allocation follows the pattern: $(0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...)$.
- Cost:
  - there are unused slots: constant-time;

  - allocation is needed: time linear in the size;
    - however *next* `append` will be constant.

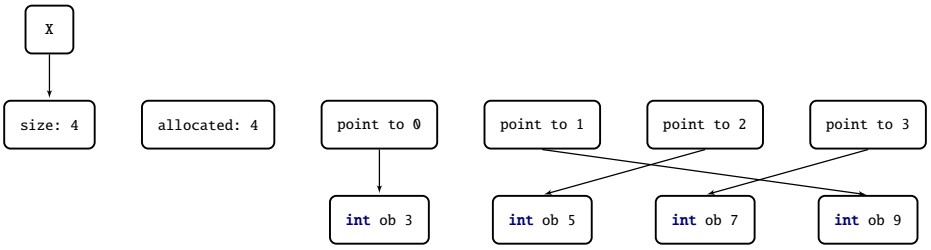
- In a long sequence of `append`, the *amortized cost* of each one is constant (contant-time op (*amortized*)).

Example 6.2.18 (**Insert inside a non empty 'list'**).
With the logic of the previous Examples, let's consider the method `insert()` on an non empty **`list`**:

- `X = [3, 5, 7]`



- `X.insert(1, 9)` ($\Longrightarrow$ `X = [3, 9, 5, 7]`)
  - shift (copy) of the pointers to make room for the new one.

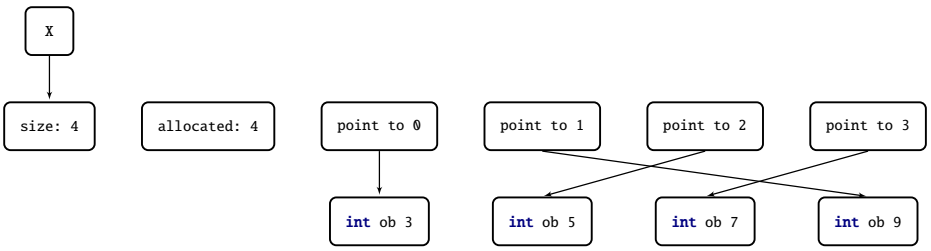


Remark 6.2.19 (**Python 'list': low dynamic of 'insert()' method**).
Considering the Example above we give here some description of the `insert()` method low dynamic:

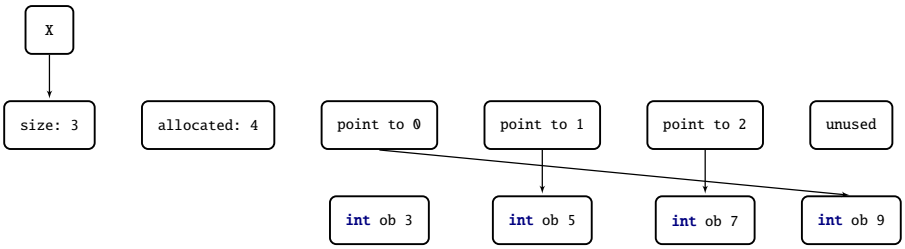
- if there is an *unused* slot for the new object (pointer): shift the pointers, use the available one;
- otherwise, over-allocate a chunk of pointers: shift the pointers, use the available one;
- cost:
  - there are unused slots: *time linear in the len*;
  - allocation is needed: *time linear in the len* + time for allocation (this will save time later).
- Linear time: $O(n)$.

Example 6.2.20 (**Low representation: 'del'**).
With the logic of the previous Examples, let's consider the method **`del`** on an non empty **`list`**:

- `X = [3, 9, 5, 7]`



- `del L[0]` ($\Longrightarrow$ `X = [9, 5, 7]`)
  - shift (copy) of the pointers.



Remark 6.2.21 (Python 'list': low dynamic of 'del' method).
Considering the Example above we give here some description of the **`del`** method low dynamic:

- shift the pointers;
- cost: *time linear in the len*;
- when size goes below some threshold, also deallocate a chunk of pointers;
- linear time: $O(n)$.

As final recap consider the following table which shows the *costs* of the main operations involving the **`list`** type ($O(1)$ is constant-time, $O(n)$ is linear-time):

Complexity of **`list`** operations		
Operation	**Complexity**	**Usage**
list creation	$O(n)$ or $O(1)$	`x = list(y)`
indexed get	$O(1)$	`a = x[i]`
indexed set	$O(1)$	`x[i] = a`
concatenate	$O(n)$	`z = x + y`
append	$O(1)$	`x.append(a)`
insert	$O(n)$	`x.insert(i, e)`
delete	$O(n)$	`del x[i]`
equality	$O(n)$	`x == y`
iterate	$O(n)$	`for a in x:`
lenght	$O(1)$	`len(x)`
membership	$O(n)$	`a in x`
sort	$O(n \cdot log(n))$	`x.sort()`

Chapter 7

Pills of NumPy

7.1 Introduction to NumPy

Definition 7.1.1 (**NumPy library**).
NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays; it essentially offers:

- true n-dimensional arrays (careful: 1-D arrays are not 1-D matrices);
- operations on those arrays: math, logical, sorting, *etc.*;
- discrete Fourier transform;
- basic linear algebra;
- basic statistics;
- basic simulation.

Definition 7.1.2 (**NumPy arrays: 'ndarray'**).
In order to utilize NumPy arrays is used `class numpy.ndarray` (always assume `import numpy as np`, hence `np.ndarray`):

- fixed size; if size changes: copying and deletion of older arrays;
- homogeneus: all elements of the same type but that type could be 'Python object' (through indirection) [it is not the intended use].

Definition 7.1.3 (**NumPy universal functions: 'ufunc'**).
Instances of `class numpy.ufunc`:

- operate in a uniform way on `ndarray`, usually *element-wise*;
- supporting array *broadcasting*,[1] type casting, other '*standard*' features;

[1]The term broadcasting describes how NumPy treats arrays with different shapes during arithmetic operations. Subject to certain constraints, the smaller array is 'broadcast' across the larger array so that they have compatible shapes.

- a `ufunc` is a '*vectorized*' wrapper[2] for a function.

Definition 7.1.4 (**NumPy main mechanisms**).
We essentially have two mechanisms:

- *vectorization*: loops are implicit;
- *broadcasting*: in certain situations the shape of an array is *adapted* to the context.

Example 7.1.5 (**Vectorization**).
Just a quick example of *vectorization*:

```
>>> import numpy as np
>>> a = np.array([1, 2, 3])
>>> a
array([1, 2, 3])
>>> b = np.array([4, 5, 6])
>>> b
array([4, 5, 6])
>>> c = a + b
>>> c
array([5, 7, 9])
>>> print(c)
[5 7 9]
```

Instead of:

```
c = []
for i in len(a):
    c.append(a[i] + b[i])
```

If `a` and `b` are multidimensional: implicit nested loops.

Example 7.1.6 (**Broadcasting**).
Just a quick example of *broadcasting*:

```
>>> import numpy as np
>>> a = np.array([1, 2, 3])
>>> d = np.array([10])
>>> e = d + a
>>> e
array([11, 12, 13])
>>> f = 10 + a
>>> f
array([11, 12, 13])
```

A dimension (an *axis*) is stretched to match the other operand.

Remark 7.1.7 (**Arrays: 'ndarray'**).
Let's analyze deeply the `class ndarray`, aliased as `array` (don't confuse with the `class array.array` of the standard lib, which is one-dimensional):

- argument is a *sequence*:

[2] In the context of software engineering, a wrapper is defined as an entity that encapsulates and hides the underlying complexity of another entity by means of well-defined interfaces.

```
>>> import numpy as np
>>> a = np.array([1, 2, 3])
# WRONG:
>>> np.array(1, 2, 3)
TypeError: array() takes from 1 to 2 positional
arguments but 3 were given
>>> np.array(1, 2)
TypeError: Cannot interpret '2' as a data type
# OK:
>>> np.array((1, 2, 3))
array([1, 2, 3])
```

- two-dimensional:

```
# 2-dim array
>>> b = np.array([[1, 2, 3], [4, 5, 6]])
>>> b
array([[1, 2, 3],
       [4, 5, 6]])
>>> print(b)
[[1 2 3]
 [4 5 6]]
# but be careful! Consider
>>> C = np.array([[2, 3],[4, 5, 6]])
__main__:1: VisibleDeprecationWarning: Creating an
ndarray from ragged nested sequences (which is a
list-or-tuple of lists-or-tuples-or ndarrays with
different lengths or shapes) is deprecated. If you
meant to do this, you must specify 'dtype=object'
when creating the ndarray.
>>> C = np.array([[2, 3],[4, 5, 6]], dtype = object)
>>> C
array([list([2, 3]), list([4, 5, 6])], dtype=object)
>>> print(C)
[list([2, 3]) list([4, 5, 6])]
```

- May declare the *type* at creation time:

```
>>> c = np.array([2, 3, 4], dtype = complex)
>>> c
array([2.+0.j, 3.+0.j, 4.+0.j])
>>> print(c)
[2.+0.j 3.+0.j 4.+0.j]
```

So, in summary for **class** `ndarray` we have:

- dimensions;
- type.

They are fixed for the entire life of the array.
Purpose: map a `ndarray` on an array of the underlying (C) machine.

Remark 7.1.8 (**Arrays: 'dtypes'**).
In the previous example we have seen that we may declare the array *type* at creation time (*e.g.* `c = np.array([2, 3, 4], dtype = complex)`):

- `dtype`: control on the representation;
- for the moment let's assume we have:
  - `int`;
  - `float`;
  - `complex`;
  - `bool`.

  They are not Python types: mapped on C type representation. `int` is not unbounded: mapped into 32 (or 64) bits two's complement integers (see in *Thonny*[3]).

Definition 7.1.9 (**Arrays: 'shape'**).
With *shape* of an array (standard PL terminology) we mean:

- number of dimensions;
- size of each dimension.

For example `C = np.array([[1, 2, 3],[4, 5, 6]])`:

- shape: $(2, 3)$.

Let's consider some methods:

- `C.shape` (it is a `tuple`);

```
>>> C.shape
(2, 3)
# but
>>> D = np.array([[2, 3], [4, 5, 6]], dtype = object)
>>> D.shape
(2,)
```

- `C.ndim`: `x.ndim` always equals to `len(x.shape)`;

```
>>> C.ndim
2
>>> D.ndim
1
```

- `C.size`: the number of elements (equivalent to `prod(x.shape)` [`prod` available from Python 3.8];

[3]Thonny is an integrated development environment for Python that is designed for beginners. It was created by Aivar Annamaa, an Estonian programmer. It supports different ways of stepping through code, step-by-step expression evaluation, detailed visualization of the call stack and a mode for explaining the concepts of references and heap.

```
>>> C.size
6
>>> D.size
2
```

- `C.dtype`: the `dtype` of each scalar object.

```
>>> C.dtype
dtype('int64')
>>> D.dtype
dtype('O')
>>> E = np.array([['a', 2, 3],[4, 5, 6]])
>>> E.dtype
dtype('<U21')
```

Definition 7.1.10 (**Creating arrays: some methods**).
Let's see also some methods capable to create arrays and the arguments of which must be a shape:

- `np.zeros((3, 2))`: **return** a zero matrix, `dtype = float64`;

```
>>> np.zeros((3, 2))
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
# but also works with []
>>> np.zeros([3, 2, 4]) # three of shape (2, 4)
array([[[0., 0., 0., 0.],
        [0., 0., 0., 0.]],

       [[0., 0., 0., 0.],
        [0., 0., 0., 0.]],

       [[0., 0., 0., 0.],
        [0., 0., 0., 0.]]])
```

- `np.ones((3, 2))`: **return** an all-one matrix, `dtype = float64`;

```
>>> np.ones((3, 2))
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

- `np.empty((3, 2))`: **return** an initialized matrix, `dtype = float64`;

```
>>> np.empty((3, 2))
array([[3.10503618e+231, 3.10503618e+231],
       [1.48219694e-323, 3.95252517e-323],
       [1.18575755e-322, 4.17201348e-309]])
```

Optional named parameter `dtype`.

Remark 7.1.11 (**Creating arrays from standard sequences**).
NumPy arrays from standard sequences '*array like*':

```
>>> y = np.array([2, 3, 1, 0])
>>> x = np.array([[1, 2.0], [0, 0], (1 + 1j, 3.)])
# note mix of 'tuple' and 'list', and types
>>> x
array([[1.+0.j, 2.+0.j],
       [0.+0.j, 0.+0.j],
       [1.+1.j, 3.+0.j]])
>>> x.dtype
dtype('complex128')
>>> type(x)
<class 'numpy.ndarray'>
```

But careful: if the shape is not correct we will get strange result:

```
np.array(([1, 2, 3, 4], [5, 6]))
__main__:1: VisibleDeprecationWarning: Creating an
ndarray from ragged nested sequences (which is a
list-or-tuple of lists-or-tuples-or ndarrays with
different lengths or shapes) is deprecated. If you meant
to do this, you must specify 'dtype=object' when creating
the ndarray.
array([list([1, 2, 3, 4]), list([5, 6])], dtype=object)
```

This is not the intended use of `ndarray`.

Definition 7.1.12 (**Creating arrays: 'arange'**).
Another important method which permits to create arrays:

- `np.arange(first, last, step)`;
  - like `range`, but `return` a 1-dim array, `dtype = int 64`; `first` and `step` are optional (same as `np.array(range(...))`.)

  ```
  >>> np.arange(10, 30, 5)
  array([10, 15, 20, 25])
  ```

- `arange` accepts `float` arguments, even a `float` `step`;
- the `dtype` of the created array object depends (more or less in the obvious way) from the types of the arguments of `arange`;
- in case of a `float` `step`, the size of the created object is not obvious, since it depends on approximations:

  ```
  >>> from numpy import pi
  >>> np.arange(pi, 3 * pi, pi)
  array([3.14159265, 6.28318531])
  # array([pi, 2pi])
  >>> np.arange(pi, 3 * pi + 10 ** -15, pi)
  array([3.14159265, 6.28318531, 9.42477796])
  # array([pi, 2pi, 3pi])
  >>> np.arange(pi, 3 * pi + 10 ** -16, pi)
  array([3.14159265, 6.28318531])
  # array([pi, 2pi])
  ```

Definition 7.1.13 (**Reshape arrays: 'reshape'**).
The method:

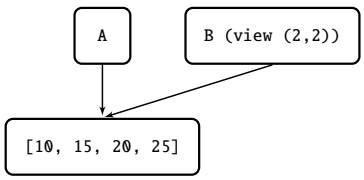
- `A.reshape(shape)`: gives *shape* to the array `A`:

```
>>> np.arange(10, 30, 5)
array([10, 15, 20, 25])
>>> np.arange(10, 30, 5).reshape((2, 2))
array([[10, 15],
       [20, 25]])
```

  `shape` must match the *size* of `A`.

- `reshape` gives a *view* on the *base array*:

```
>>> A = np.arange(10, 30, 5)
>>> B = A.reshape((2, 2))
>>> A
array([10, 15, 20, 25])
>>> B
array([[10, 15],
       [20, 25]])
>>> A[0]
10
>>> B[0]
array([10, 15])
#########################
>>> A[0] = 1
>>> A
array([ 1, 15, 20, 25])
>>> B
array([[ 1, 15],
       [20, 25]])
>>> B[0] = 2
>>> A
array([ 2,  2, 20, 25])
>>> B
array([[ 2,  2],
       [20, 25]])
```



Definition 7.1.14 (**Creating arrays: 'linespace'**).
Another important method which permits to create arrays:

- `linspace(start, end, numb)`;
  - generates an uniformly spaced array, *e.g.* `np.linspace(0, 2, 9)` generates 9 numbers from 0 to 2 (included).

```
>>> np.linspace(0, 2, 9)
array([0., 0.25, 0.5, 0.75, 1., 1.25, 1.5, 1.75, 2.])
```

Definition 7.1.15 (**Creating arrays from generators: 'fromiter'**).
An efficient way to create an array is using a generator:

```
>>> G = (i ** 2 for i in range(10))
>>> a = np.fromiter(G, dtype = int)
>>> a
array([ 0, 1, 4, 9, 16, 25, 36, 49, 64, 81])
# but pay attention
>>> a = np.fromiter(G)
TypeError: fromiter() missing required argument 'dtype'
(pos 2)
```

Definition 7.1.16 (**Creating arrays from functions: 'fromfunction'**).
Another efficient way to create an array is using a function:

```
def f(i, j):
    return i == j

def g(i, j):
    return i + j
# numpy.fromfunction(function, shape,
# dtype = <class 'float'> [by default])
>>> ba = np.fromfunction(f, (3, 3))
>>> ba
array([[ True, False, False],
       [False,  True, False],
       [False, False,  True]])
>>> ga = np.fromfunction(f, (3, 2))
>>> ga
array([[ True, False],
       [False,  True],
       [False, False]])
>>> ca = np.fromfunction(g, (3, 3))
>>> ca
array([[0., 1., 2.],
       [1., 2., 3.],
       [2., 3., 4.]])
```

Remark 7.1.17 (**Arrays: indexing**).
How does NumPy array indexing work?

- `ndarray`: indexed (element accessed) like standard Python sequences;
- n-dimensional `ndarray`: the n indexes may appear in the same `[...]` (more efficient solution).

```
>>> x = np.array([[1, 2, 3, 4],[5, 6, 7, 8]])
>>> x
array([[1, 2, 3, 4],
```

```
       [5, 6, 7, 8]])
>>> x[0][2]
3
>>> x[0, 2] # more efficient;
            # x[0][2] first construct x[0]
3
>>> x[0]
array([1, 2, 3, 4])
>>> x[0] = np.array([5, 5, 5, 5])
>>> x
array([[5, 5, 5, 5],
       [5, 6, 7, 8]])
```

Remark 7.1.18 (**Arrays: slices**).
How does NumPy array slicing work?

- Use slices like in standard Python; see some cases:

```
>>> x = np.arange(10)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> x[2: 5]
array([2, 3, 4])
>>> x[: -7]
array([0, 1, 2])
>>> x[1: 7: 2]
array([1, 3, 5])
```

And:

```
>>> y = np.arange(35).reshape(5, 7)
>>> y
array([[ 0,  1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12, 13],
       [14, 15, 16, 17, 18, 19, 20],
       [21, 22, 23, 24, 25, 26, 27],
       [28, 29, 30, 31, 32, 33, 34]])
>>> y[1: 5: 2, : : 3]

# 1: 5: 2 -> rows
# take row 1 up to 5 (excluded) with step 2

# : : 3 -> columns
# take all the columns with step 3

array([[ 7, 10, 13],
       [21, 24, 27]])
>>> y[:, 1] # the second (i.e., index 1) column of y
array([ 1,  8, 15, 22, 29])
```

Remark 7.1.19 (**Arrays: slices**).
We said in the previous Remark that we can use slices like in standard Python; but:

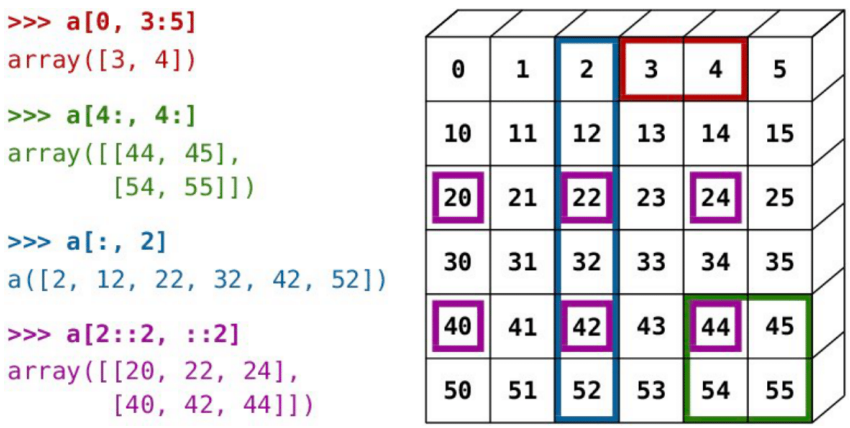


Figure 7.1: slices on arrays; visual example.

- may use slice LHS of =, but *can never make the array grow/shrink*; hence size of RHS must match size of LHS slice;
- indexing of entire dimensions and a fortiori slices *produces views of the original base array*:

```
# On standard 'list'
>>> L = [1, 2, 3, 4, 5]
>>> LL = L[1: 2]
>>> LL
[2]
>>> LL[0] = 100 # L is not changed
>>> LL
[100]
>>> L
[1, 2, 3, 4, 5]

# On 'ndarrays'
>>> X = np.array([1, 2, 3, 4, 5])
>>> XX = X[1: 2]
>>> XX
array([2])
>>> XX[0] = 100 # X is changed
>>> XX
array([100])
>>> X
array([  1, 100,   3,   4,   5])
```

- Slices produce *views*, see other examples:

```
>>> x = np.array([[1, 2, 3, 4],[5, 6, 7, 8]])
>>> x
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
>>> y = x[0]
>>> x[0, 1] = 99
```

```
>>> y
array([ 1, 99,  3,  4])
>>> y.base
array([[ 1, 99,  3,  4],
       [ 5,  6,  7,  8]])
>>> c1 = x[:, -1]
>>> c1
array([4, 8])
>>> x[-1, -1] = 88
>>> c1
array([ 4, 88])
>>> c1.base
array([[ 1, 99,  3,  4],
       [ 5,  6,  7, 88]])
```

Remark 7.1.20 (**Views**).
So, in summary:

- slices;
- entire dim selection;
- reshape.

Produces *views* on the *base array*; hence a lot of *aliasing* is around.

We have two main structural indexing tools:

- `numpy.newaxis`
- *ellipsis*: '`...`'

Let's see them.

Definition 7.1.21 (**Structural indexing tools: 'newaxis'**).
`numpy.newaxis` creates an axis of lenght one.
`newaxis` is an alias for `None`, and `None` can be used in place of this with the same result.
See some applications:

```
>>> x = np.array([1, 2, 3, 4]) # shape: (4,)
>>> x[np.newaxis, :] # shape(1, 4)
array([[1, 2, 3, 4]])
>>> x[:, np.newaxis] # shape(4, 1)
array([[1],
       [2],
       [3],
       [4]])
```

Example 7.1.22 (**NumPy 'newaxis'**).
Create a matrix of the sum of two series:

```
>>> x = np.arange(6)
>>> x
array([0, 1, 2, 3, 4, 5])
```

```
>>> y = np.arange(0, 60, 10)
>>> y
array([ 0, 10, 20, 30, 40, 50])
# T = ???
# We need 6 rows, 6 columns; then broadcasting ...
>>> y[:, np.newaxis]
array([[ 0],
       [10],
       [20],
       [30],
       [40],
       [50]])
>>> x[np.newaxis, :]
array([[0, 1, 2, 3, 4, 5]])
>>> T = y[:, np.newaxis] + x[np.newaxis, :]
>>> T
array([[ 0,  1,  2,  3,  4,  5],
       [10, 11, 12, 13, 14, 15],
       [20, 21, 22, 23, 24, 25],
       [30, 31, 32, 33, 34, 35],
       [40, 41, 42, 43, 44, 45],
       [50, 51, 52, 53, 54, 55]])
# And
>>> S = x[np.newaxis, :] + y[:, np.newaxis]
>>> S
array([[ 0,  1,  2,  3,  4,  5],
       [10, 11, 12, 13, 14, 15],
       [20, 21, 22, 23, 24, 25],
       [30, 31, 32, 33, 34, 35],
       [40, 41, 42, 43, 44, 45],
       [50, 51, 52, 53, 54, 55]])
```

General recipe for table of an operation (the 'outer operation') on two series.

Definition 7.1.23 (**Structural indexing tools: ellipsis '...'**).
Ellipsis '`...`' stands for *as many* `:` *to reach* `x.ndim`; there may only be a single ellipsis present.

```
>>> x = np.array([[[[1], [2], [3]], [[4], [5], [6]]],
    [[[1], [2], [3]], [[4], [5], [6]]]])
>>> x
array([[[[1],
         [2],
         [3]],

        [[4],
         [5],
         [6]]],


       [[[1],
         [2],
         [3]],

        [[4],
```

```
         [5],
         [6]]]]])
>>> x.shape
(2, 2, 3, 1)
>>> x.ndim
4
>>> x[0, ..., 0] # same as x[0, : , : , 0]
array([[1, 2, 3],
       [4, 5, 6]])
```

Remark 7.1.24 (**Fancy indexing**).
With the term *fancy* we mean *indirect*; specifically using *arrays as index*:

- integer arrays;
- Boolean arrays (Boolean filtering).

Warning: fancy indexing produces copies, unless it is used as LHS in an assigment.

Let's start with the integer arrays case, then we move on the Bolean arrays.

Remark 7.1.25 (**Fancy indexing: index/integers arrays**).
Indirect indexing via arrays: array of integers type may be used as indexes; the contents of the index array are the real indexes.

```
>>> x = np.arange(10, 1, -1)
>>> x
array([10,  9,  8,  7,  6,  5,  4,  3,  2])
>>> x[np.array([3, 3, 1, 8])]
array([7, 7, 9, 2])
>>> inda = np.array([3, 3, 1, 8])
>>> x[inda]
array([7, 7, 9, 2])
```

Negative values: ok as usual:

```
>>> x
array([10,  9,  8,  7,  6,  5,  4,  3,  2])
>>> x[np.array([3, 3, -3, 8])]
array([7, 7, 4, 2])
```

Index out of bounds: error (contrary to slices):

```
>>> x
array([10,  9,  8,  7,  6,  5,  4,  3,  2])
>>> x[np.array([3, 3, 20, 8])]
IndexError: index 20 is out of bounds for axis 0 with
size 9
```

Multidimensional index array: what is returned?

- same shape as the index array;
- type and values of the array being indexed.

```
>>> x
array([10,  9,  8,  7,  6,  5,  4,  3,  2])
>>> x[np.array([[1, 1], [2, 3]])]
array([[9, 9],
       [8, 7]])
```

Regarding indexing multidimensional arrays especially when also the index array is multidimensional we suggest to see the documentation (this case is not relevant for the purpose of the course).

Remark 7.1.26 (**Fancy indexing: Boolean arrays**).
If indexing array is a *Boolean array*:

- Booleans *select* the elements on the indexed array;
- `True` $\implies$ select;
- `False` $\implies$ do not select.

It is a very handy tool!

```
>>> np.random.seed(3) # to config a certain random array
>>> a = np.random.randint(0, 21, 15)
# caveat: 21 NOT INCLUDED
>>> a
array([10, 3, 8, 0, 19, 10, 11, 9, 10, 6, 0, 20, 12, 7,
       14])
>>> (a % 3 == 0) # vectorization!
array([False,  True, False,  True, False, False, False,
True, False, True,  True, False,  True, False, False])
>>> a[a % 3 == 0]
array([ 3,  0,  9,  6,  0, 12])
```

Remark 7.1.27 (**Fancy indexing: LHS**).
Above we mentioned that fancy indexing produces copies, unless used as LHS in an assigment:

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[a % 3 == 0] = 99
>>> a
array([99,  1,  2, 99,  4,  5, 99,  7,  8, 99])
b = a[a % 3 == 0]
>>> b
array([99, 99, 99, 99])
>>> a[np.array([0, 3])] = 100
>>> a
array([ 100, 1, 2,  100,   4,   5,  99,   7,   8,  99])
>>> b
array([99, 99, 99, 99])
```

Example 7.1.28 (**Boolean filtering**).
Add a constant to all negative elements:

```
>>> x = np.array([1., -1., -2., 3])
>>> x
array([ 1., -1., -2.,  3.])
>>> x[x < 0] += 20
>>> x
array([ 1., 19., 18.,  3.])
```

Definition 7.1.29 (**Axes**).
Axes: directions along dimensions.
Some methods acts on arrays (*e.g.* `a = np.array([1, 2, 3, 4])`) like `list`:

- `a.sum()`;
- `a.prod()`;
- `a.max()`;
- `a.min()`.

They may be used '*across axes*' *i.e.* along rows or columns.

```
>>> b = np.arange(12).reshape(3, 4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> b.sum(axis = 0)
# sum is taken over columns, 'along axis 0'
array([12, 15, 18, 21])
>>> b.sum(axis = 1)
# sum is taken over rows, 'along axis 1'
array([ 6, 22, 38])
```

More than 2 dimensions: '*along axis*' less intuitive. In general reduces the dimension (of 1, in this case):

```
>>> c = np.arange(12).reshape(2, 3, 2)
>>> c
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5]],

       [[ 6,  7],
        [ 8,  9],
        [10, 11]]])
>>> c.shape
(2, 3, 2)
>>> c0 = c.sum(axis = 0)
>>> c0.shape # we project along axis 0, so: (3, 2)
(3, 2)
>>> c0
array([[ 6,  8],
       [10, 12],
       [14, 16]])
>>> c1 = c.sum(axis = 1)
```

```
>>> c1
array([[ 6,  9],
       [24, 27]])
```

In summary, *NumPy axes* are the *directions* along *rows* and *columns*; just like coordinate systems, NumPy arrays also have axes.

axis 1 $\longrightarrow$

axis 0 $\downarrow$

	0	1	2	3
0				
1				
2				
3				

Definition 7.1.30 (**Boolean operations**).
Standard boolean operations are not vectorized (because they are not defined through `__xxx__` methods):

```
>>> x = np.array([[21., 19., 18., 3.], [3., 4., 19., 21.
    ]])
>>> x
array([[21., 19., 18.,  3.],
       [ 3.,  4., 19., 21.]])
>>> not (x > 4)
ValueError: The truth value of an array with more than
one element is ambiguous. Use a.any() or a.all()
>>> ~ (x > 4)
array([[False, False, False,  True],
       [ True,  True, False, False]])
```

Some functions are redefined for this. Use:

```
~ instead of not
& instead of and
| instead of or
```

NumPy re-defines boolean values:

- `np.True_`
- `np.False_`

```
>>> np.True_
True
>>> np.False_
False
```

7.2 Broadcasting, some final recipes

We already mentioned *broadcasting*, let's see it here deeply.

Remark 7.2.1 (**Broadcasting**).
Broadcasting allows functions with different shape inputs (sometimes):

```
>>> a = np.array([1]) # shape: (1,)
>>> b = np.array([7, 8, 9]) # shape: (3,)
>>> a + b
array([ 8,  9, 10]) # shape: (3,)
```

Same number of dimensions: a dimension of `1` in `a` is extended/stretched to match the corresponding dimension in `b`; the *single* value in that dimension is replicated:

- `a`: `[1]` $\longrightarrow$ `[1, 1, 1]`;
- `b`: `[7, 8, 9]`.

Another case:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]]) # shape: (2, 3)
>>> b = np.array([7, 8, 9]) # shape: (3,)
>>> a + b  # shape: (2, 3)
array([[ 8, 10, 12],
       [11, 13, 15]])
```

Different dimensions: the missing dimension in `b` is added, as `1` (like `np.newaxis`):

- `b.shape`: `(3,)` $\longrightarrow$ `(1, 3)` $\Longrightarrow$ `b`: `[7, 8, 9]` $\longrightarrow$ `[[7, 8, 9]]`;
- now we are in the '*same dimensions*': `a` $\longrightarrow$ `(2, 3)`, `b` $\longrightarrow$ `(1, 3)`;
- the *single* value in the dimension `1` is replicated to match the other:

  ```
  b: [[7, 8, 9]] -> [[7, 8, 9],
                     [7, 8, 9]]
  ```

Remember:

- shape: a `tuple (d1, d2, ..., dn)`;
- dimensions: `len(shape)` $\longrightarrow$ `n`;
- size of dimension `k`: `dk`.

Two rules of broadcasting:

1. if one argument has *less* dimensions $\Longrightarrow$ add as many leading `1s` as possible to the shape;
2. if one argument has size `1` on dimension `k` $\Longrightarrow$ stretch that dimension to match the dimension of the other array on that dimension.

After application of the rules, *the sizes of all arrays must match.*

Example 7.2.2 (Outer sum).
General recipe for table of an operation (the '*outer operation*') on two series.

```
>>> x = np.arange(6)
>>> x
array([0, 1, 2, 3, 4, 5])
>>> y = np.arange(0, 60, 10)
>>> y
array([ 0, 10, 20, 30, 40, 50])
>>> y[:, np.newaxis]
array([[ 0],
       [10],
       [20],
       [30],
       [40],
       [50]])
>>> x[np.newaxis, :]
array([[0, 1, 2, 3, 4, 5]])
>>> T = y[:, np.newaxis] + x[np.newaxis, :]
>>> T
array([[ 0,  1,  2,  3,  4,  5],
       [10, 11, 12, 13, 14, 15],
       [20, 21, 22, 23, 24, 25],
       [30, 31, 32, 33, 34, 35],
       [40, 41, 42, 43, 44, 45],
       [50, 51, 52, 53, 54, 55]])
# Could be written also as:
>>> T = y[:, np.newaxis] + x
>>> T
array([[ 0,  1,  2,  3,  4,  5],
       [10, 11, 12, 13, 14, 15],
       [20, 21, 22, 23, 24, 25],
       [30, 31, 32, 33, 34, 35],
       [40, 41, 42, 43, 44, 45],
       [50, 51, 52, 53, 54, 55]])
# But pay attention:
>>> T = y + x[np.newaxis, :]
>>> T
array([[ 0, 11, 22, 33, 44, 55]])
```

Let's analyze the process involved:

- `x.shape` $\longrightarrow$ `(6,)`:

  ```
  >>> x.shape
  (6,)
  ```

- `y.shape` $\longrightarrow$ `(6,)`:

  ```
  >>> y.shape
  (6,)
  ```

- `y[:, np.newaxis].shape` $\longrightarrow$ `(6, 1)`:

  ```
  >>> y[:, np.newaxis].shape
  (6, 1)
  # What happens is:
  ```

```
[0, 10, 20, 30, 40, 50] -> [[ 0],
                            [10],
                            [20],
                            [30],
                            [40],
                            [50]]
```

- Analyzing specifically the operation `T = y[:, np.newaxis] + x`:
  - By rule 1. of Remark 7.2.1: `x.shape` $\longrightarrow$ `(1, 6)`

    ```
    [0, 1, 2, 3, 4, 5] -> [[0, 1, 2, 3, 4, 5]]
    ```

  - Now we apply rule 2. of Remark 7.2.1 twice, once on `x` once on `y[:, np.newaxis]`:

    ```
    # x.shape -> (1, 6) -> (6, 6)

    [0, 1, 2, 3, 4, 5] -> [[0, 1, 2, 3, 4, 5]] ->

    [[0, 1, 2, 3, 4, 5],
     [0, 1, 2, 3, 4, 5],
     [0, 1, 2, 3, 4, 5],
     [0, 1, 2, 3, 4, 5],
     [0, 1, 2, 3, 4, 5],
     [0, 1, 2, 3, 4, 5]]
    ```

    ```
    # y ->  y[:, np.newaxis] -> (6, 6)

    [0, 10, 20, 30, 40, 50] -> [[ 0], ->
                                [10],
                                [20],
                                [30],
                                [40],
                                [50]]

    [[ 0,  0,  0,  0,  0,  0],
     [10, 10, 10, 10, 10, 10],
     [20, 20, 20, 20, 20, 20],
     [30, 30, 30, 30, 30, 30],
     [40, 40, 40, 40, 40, 40],
     [50, 50, 50, 50, 50, 50]]
    ```

  - In the end, just apply the *universal* (vectorized) `+`.

Example 7.2.3 (**A broadcasting exercise**).
Let's do some conceptual exercises about broadcasting:

1. Exercise:

   (a) `A` (3d array): $15 \times 3 \times 5$;

(b) `B` (3d array): $15 \times 1 \times 5$;

(c) `Result` (3d array): $15 \times 3 \times 5$.

2. Exercise:

   (a) `A` (3d array): $15 \times 3 \times 5$;

   (b) `B` (2d array): $[1 \times]3 \times 1$;

   (c) `Result` (3d array): $15 \times 3 \times 5$.

3. Exercise:

   (a) `A` (2d array): 2×1;

   (b) `B` (3d array): $8 \times 4 \times 3$;

   (c) `Result` (?d array): IMPOSSIBLE.

Let's see some additional functionalities.

Remark 7.2.4 (**Concatenate**).
How can we *concatenate* two arrays?

```
>>> Ones = np.array([[1, 1, 1], [1, 1, 1]])
>>> Ones
array([[1, 1, 1],
       [1, 1, 1]])
>>> Twos = np.array([[2, 2, 2], [2, 2, 2]])
>>> Twos
array([[2, 2, 2],
       [2, 2, 2]])
>>> Conc_over0 = np.concatenate([Ones, Twos], axis = 0)
>>> Conc_over0
array([[1, 1, 1],
       [1, 1, 1],
       [2, 2, 2],
       [2, 2, 2]])
>>> Conc_over1 = np.concatenate([Ones, Twos], axis = 1)
>>> Conc_over1
array([[1, 1, 1, 2, 2, 2],
       [1, 1, 1, 2, 2, 2]])
```

Note that in a 1-d array there is only one axis:

axis 0 $\longrightarrow$

1	1	1	1	1

An attempt to concatenate two 1-d arrays along axis 1 gives an error:
`IndexError: axis 1 out of bounds [0, 1)`.

Remark 7.2.5 (**Shape manipulation**).
All these give *views*:

- `a.ravel()`: **return** the array, flattened:[4]

[4] In `ravel` the order is 'row-major, C-style order' (the rightmost index 'changes the fastest', so the element after `a[0, 0]` is `a[0, 1]`). For use 'column-major, Fortran-style order', use `a.ravel(order = 'F')`.

```
Ones = np.array([[1, 1, 1], [1, 1, 1]])
>>> Ones
array([[1, 1, 1],
       [1, 1, 1]])
>>> Twos = np.array([[2, 2, 2], [2, 2, 2]])
>>> Twos
array([[2, 2, 2],
       [2, 2, 2]])
>>> Twos.ravel()
array([2, 2, 2, 2, 2, 2])
>>> Ones.ravel()
array([1, 1, 1, 1, 1, 1])
```

- `a.reshape(6, 2)`: modifies shape.
- `a.T`: **`return`** the array, transposed.

Definition 7.2.6 (**Transpose**).
Given an array `X` with shape `(d1, d2, ..., dn)` its *transpose* has shape `(dn, ..., d2, d1)`.
More specifically:

- `X.T` is the transpose of `X` (it is a *view* on `X`);
- `X.T[j1, ..., jn]` is `X[jn, ..., j1]`;
- Let `X` be a 1-d array and `X.shape` is `(k,)`, who is `X.T`? The same as `X`: shape `(k,)` inverted is always `(k,)`. Especially delicate if taking rows (columns) as slices of 2-d arrays:

```
>>> M = np.arange(1, 10).reshape(3, 3)
>>> M
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> M.shape
(3, 3)
>>> M[:, 0]
array([1, 4, 7])
>>> M[0, :]
array([1, 2, 3])
# same shape
>>> M[0, :].shape
(3,)
>>> M[:, 0].shape
(3,)
>>> M[0, :].T
array([1, 2, 3])
>>> M[:, 0].T
array([1, 4, 7])
```

Definition 7.2.7 (**Shape manipulation: 'resize'**).
Instead, the method `ndarray.resize` modifies the array itself:

```
>>> a = np.array([[2,8,0,6], [4,5,1,1], [8,9,3,6]])
>>> a
array([[2, 8, 0, 6],
       [4, 5, 1, 1],
       [8, 9, 3, 6]])
>>> a.reshape((2, 6))
array([[2, 8, 0, 6, 4, 5],
       [1, 1, 8, 9, 3, 6]])
>>> a
array([[2, 8, 0, 6],
       [4, 5, 1, 1],
       [8, 9, 3, 6]])
>>> a.resize((2, 6)) # equivalent a.resize(2, 6)
>>> a
array([[2, 8, 0, 6, 4, 5],
       [1, 1, 8, 9, 3, 6]])
```

Definition 7.2.8 (**Method 'random.random()'**).
The method `np.random.random()` **return** random floats on the half-open interval $[0.0, 1.0)$:

```
>>> np.random.random()
0.7637922151505848
```

In addition to this, given a shape `shape` if we use `np.random.random(shape)` it **return** an array of random numbers of the given shape:

```
>>> np.random.random((2, 3))
array([[0.5559666 , 0.64856817, 0.53544294],
       [0.58148801, 0.56450951, 0.97807617]])
```

Definition 7.2.9 (**Stacking together arrays: 'vstack' and 'hstack'**).
Several arrays can be stacked together along different axes; in particular we can use the methods:

- `np.vstac`
- `np.hstac`

Let's see an application:

```
>>> a = np.floor(10 * np.random.random((2, 2)))
>>> a
array([[4., 4.],
       [8., 2.]])
>>> b = np.floor(10 * np.random.random((2, 2)))
>>> b
array([[1., 4.],
       [0., 6.]])
>>> np.vstack((a, b))
array([[4., 4.],
       [8., 2.],
       [1., 4.],
       [0., 6.]])
```

```
>>> np.hstack((a, b))
array([[4., 4., 1., 4.],
       [8., 2., 0., 6.]])
```

Definition 7.2.10 (**Splitting arrays: 'hsplit', 'vsplit'**).
It is possible to split arrays into smaller pieces; in particular we can use the methods:

- `np.hsplit`
- `np.vsplit`

Let's see an application:

```
>>> a = np.floor(10 * np.random.random((2, 12)))
>>> a
array([[1., 1., 7., 3., 4., 2., 3., 6., 2., 0., 3., 0.],
       [0., 6., 2., 8., 2., 1., 3., 1., 6., 7., 5., 4.]])
>>> np.hsplit(a, 3) # Split a into 3
[array([[1., 1., 7., 3.],
        [0., 6., 2., 8.]]),
 array([[4., 2., 3., 6.],
        [2., 1., 3., 1.]]),
 array([[2., 0., 3., 0.],
        [6., 7., 5., 4.]])]

# Split a after the third and the fourth column
>>> np.hsplit(a, (3, 4))
[array([[1., 1., 7.],
        [0., 6., 2.]]),
 array([[3.],
        [8.]]),
 array([[4., 2., 3., 6., 2., 0., 3., 0.],
        [2., 1., 3., 1., 6., 7., 5., 4.]])]
```

In the same way `np.vsplit` splits along the vertical axis; furthermore another method `np.array_split` allows to specify along which axis to split (see the documentation).

Definition 7.2.11 (**Main diagonal index**).
It is possible to return indices to access the main diagonal of an array; in particular we can use the method:

- `numpy.diag_indices(n, ndim = 2)` with an array `a` with `a.ndim` ≥ 2 dimensions and shape `(n, n, ..., n)`.

Let's see an application:

```
>>> di = np.diag_indices(4)
>>> di
(array([0, 1, 2, 3]), array([0, 1, 2, 3]))
>>> a = np.arange(16).reshape(4, 4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
```

```
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> a[di] = 100
>>> a
array([[100,   1,   2,   3],
       [  4, 100,   6,   7],
       [  8,   9, 100,  11],
       [ 12,  13,  14, 100]])
>>> sum(a[di])
400
```

Filling the diagonal may be obtained more efficiently via:

- `numpy.fill_diagonal(a, val)`: fill the main diagonal of the given array of any dimensionality (*e.g.* `np.fill_diagonal(a, 100)`).

Remark 7.2.12 (**'fromfunction' and 'if' statement**).
We saw that `np.fromfunction` allows the construction of arrays by mapping a function (passed as argument to `np.fromfunction`) to an input array of a certain shape:

```
>>> import numpy as np
>>> def f(x, y):
      return x + y

>>> np.fromfunction(f, (3, 3), dtype = int)
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

However a program like the following returns an error:

```
>>> def f(x, y):
      if x == y:
        return 0

>>> np.fromfunction(f, (2, 2), dtype = int)
ValueError: The truth value of an array with more than
one element is ambiguous. Use a.any() or a.all()
```

The reason is *the use of the conditional command inside the function* `f` passed to `np.fromfunction`.
The implementation of `fromfunction`, in fact, applies the *vectorization* of the operations used inside its parameter (`f`).
In this way it results in a very efficient (lower level) iteration of `f` over the elements of its argument arrays.
If inside `f` we use non vectorizable operations (like the boolean operations, or the conditional `if`), we get an error:

- in `fromfunction` the guard `x == y` will evaluate:

  ```
  array([[True, False],
         [False, True]])
  ```

  Which is not a valid truth value for the guard of an `if`.

A workaround is to explicitly *vectorize* the function:

- instead of passing `f` to `fromfunction`, pass `np.vectorize(f)`:

```
>> np.fromfunction(np.vectorize(f), (1,1), dtype=int)
array([[0]])

# just note that with shape (2, 2) gives error
>> np.fromfunction(np.vectorize(f), (2,2), dtype=int)
TypeError: int() argument must be a string, a
bytes-like object or a number, not 'NoneType'

# instead with the following:
>>> def f(x, y):
    if x == y:
        return 0
    else:
        return 1

>>> np.fromfunction(np.vectorize(f), (2, 2), dtype =
    int)
array([[0, 1],
       [1, 0]])
```

`np.vectorize(f)` will use `for` loops to explicitly apply `f` to the elements of the input array, without using the built-in vectorization of the predefined operations. It is slower, of course.

Appendices

Appendix A

Sequences

Laboratory of Python

Lab 01 - Sequences

24 September 2021

Data types

Each data type consists of a set of values and a set of operators handling such values.

Possible types of data:

- integers (int)
- rational numbers (float)
- complex numbers (complex)
- boolean values (bool)
- strings (str)
- ...

Operators for integer numbers, rational numbers and complex numbers

Representations for numerical data types:

- integer: M=3
- rational (float): M=3.0
- complex: M=(3+1j)

Operators:

- M+M → sum (int, float, complex)
- M*M → product (int, float, complex)
- M/M → division (result of type float or complex)
- M//M → quotient (division quotient, result of type int)
- M%M → remainder (division remainder, only for int data type)
- M**M → exponentiation (int, float, complex)

Operations on strings

Representation for strings:

- `M = "Test"; N="home"`

Operators:

- M+N → concatenates strings M and N (ex. `Testhome`)
- M*3 → concatenates 3 times the string in M (ex. `TestTestTest`)
- len(M) → returns the length of string M (ex. `4`)
- M[0], ···, M[len(M)-1] → returns each single character of string M (ex. M[0] → `T`)

Structure of a function

```
def funcName(parameters):
    """this comment explains the
    function's implementation"""
    instructions
    ...
    instructions #brief comment
    return results
```

Function definition

- A function has got a name
- A function might have parameters (parentheses must be used anyway!)
- A colon (:) is used after a function's definition.
- A function has a content (its *body*).
  - The content of a function must be "indented": spaces must be added at the beginning of each line of the content (the body of a function has to be moved to the right of its definition in order to make the Python interpreter understand that those lines are the function's body). Conventionally **four blank spaces**, or a **tab** are used to indent a function's body.
- A function might return one or more results
- Once a function is defined ...
- ... you might use it as many times you need it.

Comments

- Can be read by a human user but are ignored by the interpreter!
- Comments are used for code documentation
- Conventionally: add a comment just below a function's definition (enclosed within two triplets of double quotes) explaining what the function does.
- Brief comments might be added besides complex instruction (starting with #)

Import module

Modules are pieces of code written to fulfill common tasks (ex. generating random numbers, performing mathematical operations, etc.)

```
from module import ...
```

This command is used to "import" the functions implemented (and saved) in the module. Some commonly used modules are:

- math:
  - contains the implementation of mathematical functions like: `sin(x)`, `cos(x)`, `sqrt(x)`, ...
  - contains the definition of mathematical constants like `e`, `pi`, ... (Neperos number and π)
- random:
  - contains functions to generate random numbers like `randint(a,b)`

Import module

- In order to use the functions defined in some module, within your program:
  - import the module before calling its functions or names, ex: **from** `math` **import** `sin`
  - use the important `sin(90)`
- To know what a module contains check its documentation, ex.: `https://docs.python.org/3/library/math.html` `https://docs.python.org/3/library/random.html`

Input

In Python, to get input from the user, you can use the input function. This function prompts the user for input and returns what he types as a string.

```
>>> input("Enter your name: ")
Enter your name: Angelo
'Angelo'
```

print and input functions are not particularly useful from the Python console, they became very useful when writing programs to be run from files.

Input and assignment

```
<var> = input("input prompt")
```

Example:

```
name = input("Enter your name: ")
```

- The user sees the message: `Enter your name:`
- The program waits for the users input until he hits the Enter button
- User's input gets stored as a **string** within the name

```
name = input("Enter your name: ")
print("Hello", name, "!")
```

Input numbers

An input from the keyboard is stored as a string. If we need a numerical input from the user we should convert its type by invoking function **int**() (for conversion to integer numbers), **float**() (for conversion to floating numbers) or **complex**() (for conversion to complex numbers).

```
age = int(input("Enter your age: "))
```

Assignment

- In Python, = **does not literally mean "is equal to"**
- We shall read it from the right to the left; for example in `x = 3+2`
  - the program evaluates (computes) what's on the right of the assignment operator (=)
  - the computed value (5, in this example) is assigned to the name `x`
  - if the name `x` was already assigned a value, the previous assignment (and value) is lost
- Thus: `x = x + 1` has a meaning for assignment!
- While `1 = x` returns an error!
  (1 is not a valid name for a variable identifier)
  the assignment operator (=) **does not satisfy the commutative property**.

Conditional Execution

```
if condition:
     if internal instructions
    ...
     if internal instructions
instructions outside the selection scope
```

- The condition (a boolean expression, type **bool** is evaluated
- **If** the boolean expression is evaluated as `True`, the internal instructions of the **if** statement are executed (notice the colon ":" and the indentation).
- If the boolean condition is `False`, the internal instructions of the **if** statement are **not** executed, and the program continues with the instructions outside the selection scope.

Relational Operators

- Here's how relational operators can be written in Python:
  - `x==y` (x is equal to y)
  - `x!=y` (x is not equal to y)
  - `x>y` (x is greater than y)
  - `x<y` (x is less than y)
  - `x>=y` (x is greater than or equal to y)
  - `x<=y` (x is less than or equal to y)
- `x==2`
- `3==x`

  The above expressions are valid syntax (given `x` is defined, both return a boolean value), while...
- `4=x`

  SyntaxError: can't assign to literal

Logical Operators

- (a **and** b) is True if **both** a **and** b are True
- (a **or** b) is True if **either or both of** a **and** b are True
- (**not** a) is True if a is False

In Logic, the operands of the logical operators should be boolean expressions, but Python is not very strict (any non-zero number is interpreted as True):

```
>>> 42 and True
True
>>> 0==False
True
```

Alternative Execution

```
if condition:
     if internal instructions
    ...
     if internal instructions
else:
     else internal instructions
    ...
     else internal instructions
 instructions outside the selection scope
```

- The condition is evaluated.
- **If** the expression is True then **only** the if internal instructions are executed.
- **Else** (the condition is False), **only** the else internal instructions are executed.
- In both cases, the program execution continues with the instructions outside the selection scope!

Example

```
def func(param):
    if param=='test':
        print('This is inside the if')
    else:
        print('This is inside the else')
        return
    print('Function main body here!!!')
    return 107
```

With conditional selection a function might follow different branches leading to different return statements: When a return statement is reached the function terminates and the program execution goes back to where the function was called!

Value `None`

`None` is a special value used in Python to mean "no value"
`None` is the value returned by functions with no return statement (or when return statements have no arguments)!

```
def f():
    x = 10
print(f())
```

- `None` is the default value for all expressions that do not return any value.
- The type of this value is `NoneType`: which contains `None` as its only element.

In conditional statements:

- `M == None` → `True` if M has been assigned the `None` value (`f()==None` is True).
- `M != None` → `True` if M has any other value **different** from `None`.

Chained Conditional

```
if condition1:
    if1 internal instr.
elif condition2:
    if2 internal instr.
    ...
elif condition3:
    if3 internal instr.
    ...
else:
    else internal instr.
```

- Use when there are more than two possible cases
- **If** condition1 is `True`, then **only** if1 internal instructions are executed.
- **Else** (if condition1 is `False`), then condition2 is evaluated (first **elif**).
- **If** condition2 is `True`, then **only** if2 internal instructions are executed.
- If condition2 is `False`), the execution moves to the following **elif**.
- and so on...
- If all conditions are `False`, and an **else** statement is given, **only** the else internal instructions are executed.

Sequences

- **Immutable**: cannot be modified after their creation
  - *strings*: sequences of characters
  - *tuples*: sequences of values separated by commas
  - *range*: sequences of integer numbers
- **Mutable**: can be modified after their creation
  - *lists*
  - *dictionaries*
  - ...

Strings

If you want to use text in Python you have to use a string. A string is created by entering text between two single or double quotation marks.

```
>>> "Python is funny"
'Python is funny'
>>> 'Is Python funny?'
'Is Python funny?'
```

When the Python console displays a string it uses single quotes. In any case: the delimiter used for a string doesn't affect how it behaves in any way.

Strings and Quotes

- One, two, or three quotes?
  - You might also use three single quotes: `'''Andrea'''`
  - and freely use single and double quotes within them: `'''He says: "l'm Andrea"'''`
  - Or three double quotes: `"""Andrea"""`
  - You might use single and double quotes within them and create newlines hitting the Return button (works both with three single and three double):

```
"""Andrea said: "Hi!"
She asked: "What's your name?"
"""
```

Operations on strings

- We have seen already the sum (+) and multiplication (*) operators applied to strings
- Relational operators can be used too (>, <, ==, ...) for lexicographical (alphabetical) ordering:

```
>>> 'Man' < 'Men'
True
```

- Strings are **immutable** (you might select sub-elements of a string but cannot reassign new values to them):

```
>>> s='test'
>>> s[0]='F'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Operations on strings

Selecting characters in a string: `[ ]`

```
Indexes from the start:  0   1   2   3   4   5
Indexes from the end:   -6  -5  -4  -3  -2  -1
                       +---+---+---+---+---+---+
            name  =    | P | y | t | h | o | n |
                       +---+---+---+---+---+---+
```

```
>>> name = 'Python'
>>> name
'Python'
>>> len(name)
6
>>> name[0]
'P'
>>> name[5]
'n'
```

```
>>> name[-1]
'n'
>>> name[-2]
'o'
>>> name[4]
'o'
>>> name[-6] + name[4]
'Po'
```

String operations

Extended *slicing* operator : `[::]`

It has the form `[begin:end:step]`.

If `step` is positive, it goes from `start` included to `end` excluded, taking one element each `step`.

If `step` is negative, we go anyway from `start` included to `end` excluded, taking one element each `step`, but suppose `start >end`. (so going backwards).

```
Indexes from the start:  0   1   2   3   4   5
Indexes from the end:   -6  -5  -4  -3  -2  -1
                       +---+---+---+---+---+---+
            name  =    | P | y | t | h | o | n |
                       +---+---+---+---+---+---+
```

String operations

From the shell:

```
>>> name[0:4:2]
'Pt'
>>> name[-5:-1:2]
'yh'
```

```
>>> name[4:0:-2]
'ot'
>>> nome[-2:-5:-2]
'ot'
```

Range

We use the function **range**() to generate sequences of integer numbers:

- **range**(n) generates the sequence of integers from 0 to $n-1$
- **range**(n,m) generates the sequence of integers from n to $m-1$
- **range**(n,m,s) generates the sequence of integers from n to $m-1$ with the given step s

Note that:

- All parameters must be integers
- All parameters can be positive or negative

Example

We use the function **range**() to generate sequences of integer numbers:

- **range**(3) $= < 0, 1, 2 >$
- **range**(2,7) $= < 2, 3, 4, 5, 6 >$
- **range**(0,10,2) $= < 0, 2, 4, 6, 8 >$
- **range**(10,6) $= <>$
- **range**(10,6,-1) $= < 10, 9, 8, 7 >$
- **range**(-7,-1,2) $= < -7, -5, -3 >$

Operator in

It is possible to use **in** and **not in** to know if an element belongs (or not) to a sequence. Check the result of the following (boolean) expressions:

- `'h' in 'Python'`?
- `3 in range(2,9)`?
- `'p' not in 'Python'`?
- `'yth' in 'Python'`? (**in** can match sub-strings in strings)

Within the `string` module a few constants are pre-defined :
`import string`

- `string.ascii_letters`
- `string.ascii_lowercase`
- `string.ascii_uppercase`
- `string.digits`
- `string.punctuation`
- `string.whitespace`

- `'3' in string.digits`?

for loops

```
for <var> in <seq>:
    <instructions>
```

- `seq`: a sequence (ex. string, range...)
- `var`: name of the variable that assumes, one by one, in sequence, the values of all the elements in `seq`
- `instructions`: set of instructions to be executed cyclically **for each element** in the sequence

Tuples

- **Ordered sequences** of values (separated by commas):
  `data = 'Anne', 'Smith', 'F', 20`
- Parentheses are optional but commonly used: The IDLE shell uses parentheses when formatting tuples:

```
>>> data
('Anne', 'Smith', 'F', 20)
```

- Commas are mandatory:
  - `('Michael',)` is a tuple (with just one element)
  - while `('Michael')` is a string!
  - Notice: `()` is an empty tuple!

```
>>> type(())
<class 'tuple'>
```

- The sum (+) operator can be used to concatenate tuples.
- Selection (`[]`) and slicing (`[:]`, `[::]`) operators can be applied to tuples in the usual way:
  - What is `data + ('August',)`
  - What is `data[2]`?

Tuples and assignment

- Values can be packed within a tuple labelled with a single variable:
  `data = ('Anne', 'Smith', 'F', 20)`
- or we can unpack a tuple in a congruous number of variables:
  `(name, surname, sex, age) = data`

Parentheses are optional in both cases! In our example we will have:

```
>>> name
'Anne'
>>> surname
'Smith'
>>> sex
'F'
>>> age
20
>>> data
('Anne', 'Smith', 'F', 20)
```

Ranges and tuples

We know ranges are immutable sequences representing intervals of integer numbers:

- **range**(n) is the interval on integer numbers $[0, n[$
- **range**(a,b) is the interval on integer numbers $[a, b[$
- **range**(a, b, s), if $s > 0$, is the interval on integer numbers $[a, a+s, a+2s, ..., a+is]$ with $a+is < b$
- **range**(a, b, -s), if $s > 0$, is the interval on integer numbers $[a, a-s, a-2s, ..., a-is]$ with $a-is > b$

To visualize the content of a range we can convert it to a tuple:

```
>>> range(10)
range(0, 10)
>>> tuple(range(10))
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> tuple(range(0,30,5))
(0, 5, 10, 15, 20, 25)
```

```
>>> tuple(range(10,1,-2))
(10, 8, 6, 4, 2)
>>> tuple(range(0))
()
>>> tuple(range(10,0))
()
```

Functions (methods) on strings

In the exercises, you will need to use some functions on strings. In particular

```
>>> str.upper("Ehi!")
"EHI!"
>>> str.lower("Ehi!")
"ehi!"
```

Functions can be found here: `https://docs.python.org/3/library/stdtypes.html#string-methods`

Actually, these are **methods**, you will learn soon to more conveniently write

```
>>> "Ehi!".upper()
"EHI!"
>>> "Ehi!".lower()
"ehi!"
```

Appendix B

while and *lists*

Python lab

Lab 2 - while and lists

30 September 2021

While

while

```
while condition:
    instruction inside while
    ...
    instruction inside while
instructions after while
```

- The `condition` (Boolean expression) is evaluated.
- **Only if** the Boolean expression is `True` then the instructions within the `while` are executed (note the indentation).
- When you finish the instructions inside the while, **you go back to test the** `condition`
- If it is still `True`, you run the instructions inside the while again.
- It continues like this until the condition becomes `False`.
- If the condition is `False` you move on to the instructions after `while`.

while

```
while condition:
    inside while
```

- We can think of it as *repeat while the condition remains true*.
- Attention: it is necessary to verify that the expression present in the condition is modified in the instructions inside the while.
- Even though the shape is visually similar, the construct is very different semantically from the construct `if`.
- What happens when running this program?

```
a = 3
while a>0:
    print("Positive number")
```

Lists

List: definition and operators

Ordered and **mutable** sequences of elements.

- Elements within square brackets [1,2,3]
- Empty list: []
- List with only one item: [42]

Operations common to all sequences (already seen):

- Operator + to concatenate lists
- Operator * to repeat items (e.g. [1]*5 = ?)
- Operator [ ] to select individual elements (e.g. [1,2,3][0] = ?)
- Slicing Operator [:] and [::]
- Operator **in** and **not in**
- Function `len()`.
- Cycle **for** to iterate on the elements of the list

List: sequences mutable

Being mutable sequences they allow other operations.

- Assignment: `L[i] = 3`
- Assignment to a sublist: `L[1:3] = [28, 30]`
- Insertion of a sublist:

```
>>> L = ["a", "d", "f"]
>>> L[1:1] = ["b", "c"]
>>> L
['a', 'b', 'c', 'd', 'f']
```

- Deletion of an item: **del** `L[i]` or of sublist: **del** `L[1:5]`
- Insert an element at the bottom of a list: `L.append(x)`
- ... but also `L += [x]`
- ... but also `L = L + [x]` (which however creates a copy...)
- Insert an item in a specific position `i`: `L.insert(s, x)`

List methods

What does this program print?

```
L = [1,2,3]
L = L.append(4)
print(L)
```

Many methods on lists, including `append`, `insert`, `extend`, `clear` modify the list (but do not change the identity) and return `None`.

What does `print(L.append(4))` print?

List: aliasing vs copies

- Objects and values:

```
>>> X = [1, 2, 3]
>>> Y = [1, 2, 3]
>>> X == Y
True
>> X is Y
False
```

- Create a copy of a list. Difference between:

```
A = [1, 2, 3]
B = A
```

e

```
A = [1, 2, 3]
B = A.copy() #equivalent: b = a[:]
```

- What is `B` referring to in the first case? And in the second?
  Click here to try on PythonTutor

Appendix C

Dictionaries and comprehesion

Python lab

Lab 3 - dictionaries and comprehension

7 October 2021

Dictionaries

Definition and Initialization

Dictionaries:

- **mutable**.
- unlike lists, where the elements have a numeric key, here **the key can be of any type *immutable***.
- **elements are NOT sorted** (or, from Python 3.7, sorted by input - in any case, we will always use them as if they were NOT sorted).
- lab-info-mat runs Python 3.5, hence there are no guarantees of ordering, and you could have different results in Thonny and in lab-info-mat. → do not use the "ordering" of dictionary as a tool for solving exercises

To initialize create an empty dictionary:

```
D = {}
```

Definition and Assignment

It is a set of pairs key:value (called *item*)

```
D = {key:value, key2:value2, ...}
```

key can be string type, whole, tuple, etc... any immutable type

value can be of any type

For example:

```
D =  { 'vowels' : [ 'a', 'e', 'i', 'o', 'u' ], '
    consonants' : [ 'b', 'c' ], 'punctuation' : [':',
    ';'] }
```

keys in this case they are: `'vowels'`, `'consonants'` , `'punctuation'`

values referenced by the respective keys are `['a','e','i','o','u']`, `['b','c']`, `[',',';']`

Dictionary operations

Access `D['punctuation']` →`[',',';']` If the key is not present, error.

Edit `D['vowels'] = ['a','e','i','o','u', 'A','E','I','O' ,'U']` → modifies the key already present with the values to the right of the equals

Assignment `D['alphabet']=['a','b', ... ]` → insert in dictionary *D* a new item: *chiave* associated with values on the right hand side of the =

Deletion **del** `D['consonants']` → deletes from *D* the key and the associated values

Dimention **len**`(D)` → counts the number of items (pairs key:value) in the dictionary *D*

Methods and iteration

keys `D.keys()` → returns the *dynamic view* of the keys
N.B! It is not (yet) a list, but we can generate it with **list** `(D.keys())` or even **list** `(D)`

values `D.values()` → returns the dynamic view of the values.

item `D.items()` → returns the dynamic view of the items...

iteration **for** `k` **in** `d` itera on the keys...

presence `k` **in** `d` returns `True` if `k` **in** `D.keys()`

access For convenience, `D.get(k,default)` returns the index element in `k` in `d`! (that is `D[k]`), if it exists (`k` **in** `d`), or the value `default`. (if we do not pass the second parameter, `default=None`).

update `D1.update(D2)` extends D1 with the elements of D2, updating D1 with the values of D2 if there are equal keys in D1 and D2

Comprehension

List Comprehension

The *list comprehension*, inspired by the mathematical notation of set construction, is a useful way to quickly create lists whose content respects simple rules.

For example the analogue of

$$\{x^3 \mid x \in [0..4]\}$$

could be

```
cubes = [x**3 for x in range(5)].
print(cubes)
```

```
>>>
[0, 1, 8, 27, 64]
>>>
```

List Comprehension with if

A comprehension list can contain a **if** to filter only those values that meet a certain condition into the resulting list.

For example, the analog of

$$\{i^2 \mid i \in [0..9] \wedge i \text{ is even}\}$$

could be

```
evensquared = [i**2 for i in range(10) if i % 2 == 0]
print(evensquared)
```

```
>>>
[0, 4, 16, 36, 64]
>>>
```

Example

Return the list of the vowels in a string.

```
def vowels(s):
    RES = []
    for c in s:
        if c in "aeiouAEIOU":
            RES.append(c)
    return RES
```

```
def vowels_c(s):
    return [c for c in s if c in "aeiouAEIOU"]
```

List Comprehension: general syntax i

A comprehension list consists of brackets (squares) containing an *expression* followed by a `for` loop, then **zero or more** `for`s and/or `if`s.

The result will be a new list resulting from the evaluation of the expression in the context of the following `for` and `if`. For example, the following comprehension combines the elements of two lists if they are not the same:

```
>>> comb = [(x, y) for x in [1,2,3] for y in [3,1,4]
    if x != y]
>>> print(comb)
[(1,3), (1,4), (2,3), (2,1), (2,4), (3,1), (3,4)]
```

List Comprehension: general syntax ii

and is equivalent to:

```
comb = []
for x in [1,2,3]:
    for y in [3,1,4]:
        if x != y:
            comb.append((x, y))
```

and conceptually similar to

$$\{(x, y) \mid x \in [1, 2, 3] \wedge y \in [3, 1, 4] \wedge x \neq y\}$$

If the expression is a tuple, the parentheses are mandatory!

List Comprehension: Examples i

```
>>> L = [-4, -2, 0, 2, 4]

# create a new list with values multiplied by 2
>>> [x*2 for x in L].
[-8, -4, 0, 4, 8]

# create a list with only non-negative values
>>> [x for x in L if x >= 0]
[0, 2, 4]

# apply a function to each element of a sequence
>>> [abs(x) for x in L].
[4, 2, 0, 2, 4]
```

List Comprehension: Examples ii

```
# call a method on each element of a sequence
>>> fruit = ['BANANA', 'BLUEBERRY', 'PASSIONFRUIT']
>>> print([fruit.lower() for fruit in fruit])
['banana', 'blueberry', 'passionfruit']

# create a list of tuples (element, square) of the
    first 6 natural numbers
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]

# remember: the tuples go in parentheses
>>> [x, x**2 for x in range(6)].
SyntaxError: invalid syntax
```

List Comprehension: Examples iii

```
#print a list of all letters
>> from string import ascii_letters
>>> print([c for c in ascii_letters])
['a', 'b', 'c', ..., 'z', 'A', 'B', ..., 'Y', 'Z']

#Print the coordinates of the battleship board game...
>> from string import ascii_uppercase
>>> print([(ascii_uppercase[i], i+1) for i in range
    (10)])
('A', 1), ('B', 2), ('C', 3), ('D', 4), ('E', 5), ('F'
    , 6), ('G', 7), ('H', 8), ('I', 9), ('J', 10))]

#... alternative with zip
>>> print([(x,y) for x,y in zip(ascii_uppercase,range
    (1,11)])
```

Nested List Comprehension

The sequence to which the rules of a comprehension apply may itself be a list comprehension

```
l = ['*'+s+'*' for s in [c.upper() for c in 'abcd']]
print(l)
```

```
['*A*', '*B*', '*C*', '*D*']
```

The initial expression of a list comprehension can be a list comprehension

```
double = [[c*2 for c in word] for word in ['Da', 'casa
    ']]
print(double)
```

```
[['DD', 'aa'], ['cc', 'aa', 'ss', 'aa']]
```

Dictionary Comprehension

If we use curly brackets, and the expression is a couple `key:value`, we build a dictionary.

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

```
>>> l = ['this', 'class', 'ago', 'confusion']
>>> {word: len(word) for word in l}
{'this': 6, 'class': 6, 'ago': 2, 'confusion': 10}.
```

```
>>> {c: c.upper() for c in 'letters'}
{'l': 'L', 'e': 'E', 't': 'T', 'r': 'R'}
```

Generators (hints)

Putting instead the expressions in the notation seen so far in round brackets does **not** create tuples, but **generators**.

```
>>> p = (i for i in range(100) if i%2 == 0)
>> type(p)
<class 'generator'>
```

This notation can also be used within functions that expect sequences or *iterators*, for example if I want to calculate

$$\sum_{i=0}^{20}(i \text{ s.t. } i \text{ is not divided by } 6)$$

I can write

```
>>> sum(i for i in range(21) if i % 6 != 0)
154
```

Appendix D

OOP: classes, attributes, methods, magic methods

Python lab

Lab 4 - OOP: classes, attributes, methods, magic methods

8 October 2021

Object Oriented Programming

Object Oriented Programming

Python provides features that support object-oriented programming (OOP):

- In OOP the focus is on the creation of objects which contain both data (**attributes**) and functionality (**methods**).
- An object is an *instance* of a **class**; Classes describe what the objects will be (a class is an object's specification, or definition).
- You can use the same class as a blueprint for creating multiple different objects.
- Classes are created using the keyword **class** and an indented block, which contains the class attributes (data) and methods (which are like functions). Ex.:

```
class className:
    <Attributes>
    <Methods>
```

Initialization

The `__init__` method is called when, using the class name as a function, an instance (an object) of the class is created.

- All methods must have **self** as their first parameter but you do not need to include it when you call the methods. **Within a method definition, self refers to the instance calling the method**.
- In the `__init__` method, **self** is used to set the **initial values** of an instance's fundamental attributes.

```
class Dog:
  def __init__(self, name, color):
    self.name = name
    self.color = color

dog_instance = Dog('Fido', 'brown')
```

Attributes

- Instances of a class have **attributes**: pieces of data associated with them.
- Attributes can be accessed by putting a dot and the attribute name after an instance.

```
class Dog:
  legs = 4 # class attribute
  def __init__(self, name, color):
    self.name = name
    self.color = color

dog1 = Dog('Fido', 'brown')
dog2 = Dog('Spotty', 'white')

print("The dog is called", dog1.name)
print("The dog has color", dog2.color)
```

Class Attributes

- Classes can also have class attributes, created by assigning variables within the body of the class.
- The value is the same for all the instances of a class, and can be modified
- Class attributes can be accessed usually by putting a dot and the attribute name after the name of the class, both inside and outside the class.

```
class Dog:
  legs = 4 # class attribute
  def __init__(self, name, color):
    self.name = name
    self.color = color

dog1 = Dog('Fido', 'brown')

print("All dogs have", Dog.legs, "gambe")
```

Class Attributes: attention

```
class Dog:
  legs = 4 # class attribute
  def __init__(self, name, color):
    self.name = name
    self.color = color

dog = Dog('Fido', 'brown')
print(dog.legs) # accessing class attribute in a non
    standard way
dog.legs = 3 # creating a new instance attribute
""" Disclaimer: no animals were harmed
              by the execution of this code """
print(dog.legs) # instance shadows class attribute
print(Dog.legs) # accessing class attribute
```

See on PythonTutor

Class Methods

- Classes can have methods defined to add functionality to them. They express an action that is possible on that object.

A method differs from a function in two aspects:

- It belongs to a class (and it is defined within a class)
- The first parameter in the definition of a method has to be a reference to the instance which called the method. This parameter is usually called **self**.[1]

Methods are accessed using the same dot syntax as attributes.

[1] Technically, the parameter name **self** is just a convention: it could be changed to anything else. However, this convention is universally followed: it is wise to stick to it.

Class Methods: Example

```
class Dog:
  def __init__(self, name, color):
    self.name = name
    self.color = color
  def bark(self):
    print('Woof!')

fido = Dog('Fido', 'brown')
fido.bark()
```

```
>>>
Woof!
>>>
```

Class Methods: Example (2)

```
class Robot:
  def __init__(self, name = None):
      self.name = name
  def say_hi(self):
      if self.name:
          print('Hi, I am ' + self.name)
      else:
          print('Hi, I am a robot without a name')

x = Robot()
x.say_hi()
y = Robot('Marvin')
y.say_hi()
```

```
>>>
Hi, I am a robot without a name
Hi, I am Marvin
>>>
```

Private Attributes and Methods

- Private attributes and methods can only be accessed and modified / called from within a class
- Private attributes and methods might be used to restrict external access to data and functionalities of a class
- Attributes and methods are made private by prefixing their names with a **double underscore**:

Private Attributes and Methods

```
class Spam:
    __egg = 8 # private attribute
    def print_egg(self): # public method
        self.__increase()
        print(self.__egg)
    def __increase(self): # private method
        self.__egg += 1

s = Spam() # new instance of class Spam
```

```
>>> s.print_egg()
9
>>> print(s.__egg)
AttributeError: 'Spam' object has no attribute '__egg'
>>> s.__increase()
AttributeError: 'Spam' object has no attribute '__increase'
```

"Private" Attributes and Methods

Note however, attributes and methods are not really private:

```
print(s._Spam__egg)
s._Spam__increase()
print(s._Spam__egg)
```

```
9
10
```

Magic methods

Methods (staring and finishing with a __, a *double under* or *dunder*) that, if implemented, provide special functionalities for the class. Just a few examples:

- `__eq__(self, other)` Defines behavior for the equality operator, ==.
- `__add__(self, other)` Implements addition.
- `__sub__(self, other)` Implements subtraction.
- `__mul__(self, other)` Implements multiplication.
- `__floordiv__(self, other)` Implements integer division using the // operator.
- `__div__(self, other)` Implements division using the / operator.
- `__str__(self)` Defines behavior for when str() is called on an instance of your class. Returns a "nice", "human", "readable" string representation of the object.
- `__repr__(self)` Returns a string that shoul correspond to the Python istruction useful for creating that instance.
- `__len__(self)` Returns the length of the object

Is an object instance of a class?

We can use the special function **isinstance** to determine if an object is an instance of a certain class

```
d = Dog('Fido', 'brown')
c = Cat('Lulu', 'grey')

isinstance(d,Dog) #True
isinstance(c,Dog) #False

```

Appendix E

OOP: inheritance; exceptions

Python lab

Lab 5 - OOP: iheritance; exceptions

14 October 2021

Inheritance

Inheritance

- Inheritance provides a way to share functionality between classes.
  - Imagine several classes, `Cat`, `Dog`, `Rabbit` and so on. Although they may differ in some ways (only `Dog` might have the method `bark`), they are likely to be similar in others (all having the attributes `color` and `name`).
  - This similarity can be expressed by making them all inherit from a superclass `Animal`, which contains the shared functionalities.
- A class that inherits from another class is called a **subclass**.
- A class that is inherited from is called a **superclass**.
- To inherit a class from another class, put the superclass name in parentheses after the class name.
- If a subclass has attributes or methos with the same name of ones in the superclass, subclass overrides them.
- One class can inherit from another, and that class can inherit from a third class.
- Private attributes (starting with `__`) are not visible in subclasses.

Inheritance: Example

```
class Animal: # superclass
  def __init__(self, name, color): # shared functionality
    self.name = name
    self.color = color

class Dog(Animal): # subclass
  def bark(self):
    print('Woof!')

class Cat(Animal): # subclass
  def purr(self):
    print('Purr...')

fido = Dog('Fido', 'brown')
fido.bark()
ginger = Cat('Ginger', 'red')
ginger.purr()
```

Inheritance: Example (2)

```
class Animal: # superclass
  def __init__(self, name, color):
    self.name = name
    self.color = color

class Dog(Animal): # subclass and superclass
  def bark(self):
    print('Woof!')

class Wolf(Dog): # subclass
  def bark(self): # method overriding
    print('Grrr...')
```

Inheritance: dynamic selection of methods

```
>>> fido = Dog('Fido', 'brown')
>>> fido.bark()
Woof!
>>> sharp = Wolf('Sharp Tooth', 'grey')
>>> sharp.bark()
Grrr...
```

Is an object instance of a class/superclass?

We can use the special function **isinstance** to determine if an object is an instance of a certain class and/or its superclasses.

On the other hand, **type** returns only the specific class (the type)

```
>>> sharp = Wolf('Sharp Tooth', 'grey')
>>> type(sharp)
<class '__main__.Wolf'>
>>> isinstance(sharp,Wolf)
True
>>> isinstance(sharp,Dog)
True
>>> type(sharp) is Dog
False
>>> isinstance(sharp,Animal)
True
>>> isinstance(sharp,Cat)
False
```

The super() function i

We can use the **super**() function to "read" the `self` object as if it were an instance of its superclass.

In the following example, within the class `Cat`, **super**().__init__() refers to the `__init__` of `Cat` (which in turn inherits the `__init__` of `Animal`).

The call **super**().meow() refers to the `meow()` of `Cat`.

The super() function ii

```
class Animal: # superclass
  def __init__(self, name, color):
    self.name = name
    self.color = color

class Cat(Animal):
  def meow(self):
    print('Miao!')

class Siamese_Cat(Cat):
    def __init__(self, name):
        super().__init__(name, "white-cream-black")
        self.pedigree = True
    def meow(self):
        for i in range(3):
            super().meow()
```

Exceptions

Exceptions

- Are *raised* when a run-time/dynamic error happens
- Can be caught with the **try**...**except** construct
- Both pre-existng and user-defined exceptions can be raised with the **raise** command
- User-defined are subclasses of `Exception` class

Try... except... else... finally

```
try:
    <block> # run this first
except name1:
    <block> # run if name1 is raised in try block
except (name2, name3):
    <block> # run if any of these two exceptions raised in try
except name4 as name:
    <block> # run if name4 is raised, assign instance to name (to access the specific string value of that exception)
except:
    <block> # run for all other exceptions raised
else:
    <block> # run if no exception was raised in try block
finally:
    <block> # executed anyway before leaving the try
```

Rasing exeptions

- Exceptions are raised by the Python machine when something happens (e.g. `1/0` will raise the `ZeroDivisionError` exception)
- We can deliberately raise exeptions with the **raise** command
- We can raise both pre-defined and user-defined exceptions

A comprehension exercise

What will the following code print?

```
class MyException(Exception): pass

def f(x):
    if x < 0:
        raise MyException
    if type(x) != int:
        raise TypeError
    print(x)

for v in [-3, 2.5, 0]:
    try:
        f(v)
    except MyException:
        print("Insert a positive number")
    except TypeError:
        print("Insert an integer")
```

Appendix F

NumPy

Python lab

Lab 6 - NumPy

15 October 2021

NumPy

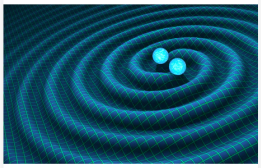
NumPy

- Numpy is the core library for scientific computing in Python.
- It provides a high-performance multidimensional array object, and tools for working with these arrays.
- If you are already familiar with MATLAB, you might find this tutorial useful to get started with Numpy.
- It is at the basis of top notch scientific computations

FIRST IMAGE OF A BLACK HOLE

How NumPy, together with libraries like SciPy and Matplotlib that depend on NumPy, enabled the Event Horizon Telescope to produce the first ever image of a black hole

DETECTION OF GRAVITATIONAL WAVES



In 1916, Albert Einstein predicted gravitational waves; 100 years later their existence was confirmed by LIGO scientists using NumPy.

SPORTS ANALYTICS

Cricket Analytics is changing the game by improving player and team performance through statistical modelling and predictive analytics. NumPy enables many of these analyses.

Arrays

- A numpy array is a grid of values, all of the same type, and is **indexed by a tuple of nonnegative integers**.
- The number of dimensions is the *rank* of the array;
- The **shape of an array is a tuple of integers** giving the size of the array along each dimension.
- We can initialize numpy arrays from nested Python lists, and access elements using square brackets.

Arrays

```
import numpy as np

a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))  # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5        # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]]) # Create a rank 2 arr.
print(b.shape)                     # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

Creating arrays

```
a = np.zeros((2,2))    # Create an array of all zeros
print(a)               # Prints "[[ 0.  0.]
                       #          [ 0.  0.]]"
print(type(a[0, 0]))   #<class 'numpy.float64'>

b = np.ones((1,2))     # Create an array of all ones
print(b)               # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7)   # Create a constant array
print(c)                # Prints "[[ 7.  7.]
                        #          [ 7.  7.]]"

d = np.eye(2)          # Create a 2x2 identity matrix
print(d)               # Prints "[[ 1.  0.]
                       #          [ 0.  1.]]"

e = np.random.random((2,2))  # Array w. random val.
print(e)                     # Who knows??
```

Creating arrays

- With a range (ints or floats) `np.arange(first,last,step)`
- Random range: `np.random.randint(first,last,number)`
- With a function

```
def g(i):
    return i % 10

a = np.fromfunction(g,(20,))
#what is a?
```

NB: you will be in trouble if you need to use a boolean or an **if** inside g...

Datatypes

- Every numpy array is a grid of elements of the same type.
- Numpy provides a large set of numeric datatypes that you can use to construct arrays.
- Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype.

```
>>> np.zeros((2,2))
array([[0., 0.],
       [0., 0.]])
>>> np.zeros((2,2), dtype=int)
array([[0, 0],
       [0, 0]])
```

You can read all about numpy datatypes in the documentation.

Reshaping arrays

- `A.reshape(shape)`, shape is a tuple, must match the size of A, returns a view on A

```
>>> np.random.randint(1,10,100).reshape((10,10))
array([[5, 2, 8, 6, 4, 7, 8, 8, 6, 3],
       [9, 8, 8, 8, 9, 3, 2, 5, 6, 8],
       [3, 7, 6, 9, 9, 3, 7, 4, 7, 1],
       [2, 2, 1, 6, 8, 7, 3, 3, 8, 8],
       [3, 8, 6, 8, 7, 1, 6, 2, 4, 4],
       [5, 9, 9, 9, 7, 6, 7, 1, 1, 1],
       [8, 2, 8, 7, 4, 5, 5, 8, 6, 7],
       [7, 4, 2, 7, 4, 3, 2, 2, 7, 5],
       [9, 7, 8, 8, 2, 8, 7, 4, 8, 1],
       [7, 7, 5, 7, 7, 3, 4, 2, 8, 5]])
```

Array indexing

Numpy offers several ways to index into arrays.

Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array

```
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

b = a[:2, 1:3]
# [[2 3]
#  [6 7]]

# A slice of an array is a view into the same data, so modifying it will modify the original array.
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is a[0, 1]
print(a[0, 1])   # Prints "77"
```

Mixing indexing

You can also mix integer indexing with slice indexing.

However, doing so will yield an array of lower rank than the original array.

```
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Mixing integer indexing with slices yields an array of lower rank, while using only slices yields an array of the same rank as the original array:
row_r1 = a[1, :] #Rank 1 view of the second row of a
row_r2 = a[1:2, :] #Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # "[[5 6 7 8]] (1, 4)"

# Same for columns
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape) #[ 2  6 10] (3,)
print(col_r2, col_r2.shape) #[[ 2],[ 6],[10]] (3, 1)
```

Integer array indexing

- Slicing returns views, array indexing creates copies (unless is used on the left of a =)

```
a = nparray([18,10,17,15,10,16,18,17,17,11])
b = np.array([0, 2, 0, 1])

#Select element from each row of a using b indexes
c = a[b]
#"c now is array([18, 17, 18, 10])"
# c is not linked with a (check c.base)

# Mutate one element from each row of a using the indices in b
a[b] -= 30
# a now is modified as
# array([-12, -20, -13,  15,  10,  16,  18,  17,  17, 11])
```

Example: selecting and permuting columns

```
import numpy as np

a = np.arange(10,20).reshape((2,5))
"""
[[10 11 12 13 14]
 [15 16 17 18 19]]
"""

b =  np.array([3,2,0,4,4])

print(a[:, b])
"""
[[13 12 10 14 14]
 [18 17 15 19 19]]
"""
```

Boolean array indexing

Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition.

```
a = np.array([[1,2], [3, 4], [5, 6]])
"""Find the elements of a that are bigger than 2;
this returns a numpy array of Booleans of the same
shape as a, where each slot of bool_idx tells
whether that element of a is > 2."""
bool_idx = (a > 2)   #  [[False False]
                     #    [ True  True]
                     #    [ True  True]]
'''We use boolean array indexing to construct a rank 1
    array consisting of the elements of a
   corresponding to the True values of bool_idx'''
print(a[bool_idx])  # [3 4 5 6]
# We can do all in a single concise statement:
print(a[a > 2])     # Prints "[3 4 5 6]"
```

Example

Write a NumPy program to compute an element-wise indication of the sign for all elements in a given array.

```
import numpy as np
x = np.array([1, 3, 5, 0, -1, -7, 0, 5])
x[x>0] = 1
x[x<0] = -1
#x is modified to
#array([ 1,  1,  1,  0, -1, -1,  0,  1])
```

For brevity we have left out a lot of details about numpy array indexing; if you want to know more you should read the documentation.

Newaxis

`np.newaxis` create an axis of length one

```
>>> x=np.array([1,2,3,4]) shape: (4,)
>>> x[np.newaxis,:]
array([[1, 2, 3, 4]]) shape: (1,4)
>>> x[np.newaxis,:].shape
(1, 4)
>>> x[:,np.newaxis] shape: (4,1)
array([[1],
[2],
[3],
[4]])
```

Broadcasting

Powerful mechanism that allows to work with arrays of different shapes when performing arithmetic operations.

```
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11,
    12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)  # Prints "[[ 2  2  4]
          #           [ 5  5  7]
          #           [ 8  8 10]
          #           [11 11 13]]"
```

1. if one arguments has less dimensions add as many 1s as possible to the shape
2. if one argument has size 1 on dimension k stretch that dimension to match the dimension of the other array on that dimension

Universal functions

- Functions that support broadcasting are known as universal functions.
- Universal functions operate in a uniform way on ndarrays, usually element-wise
- You can find the list of all universal functions in the documentation.

Axes

- Axes: directions along dimensions
- Some methods acts on arrays like lists, for example:
  a.sum()
  a.prod()
  a.max()
  a.min()

```
x = np.array([[1,2],[3,4]])

print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

Bibliography

[1] John V. Guttag, *Introduction to Computation and Programming Using Python (Second Edition: With Application to Understanding Data)*, MIT Press, 2016.

[2] Allen B. Downey, *Think Python: How to Think Like a Computer Scientist, 2nd Edition*, O'Reilly Media, 2016.

`https://greenteapress.com/wp/think-python-2e/`.

[3] Jessen Havill, *Discovering Computer Science: Interdisciplinary Problems, Principles, and Python Programming*, Chapman and Hall/CRC, 2015.