Advanced Python for advanced users

… And a few concepts of computer science

Pierre Beaujean (pierre.beaujean@unamur.be)

March 2025 (version of April 7, 2025)

University of Namur

Table of content

Back to basics

Computer?

As far as you are concerned, a computer contains:

- A **CPU**, which execute (*assembler*) code. Nowadays, there are also **GPUs** which can fulfill this role.
- Different kind of **memories** (cache, RAM, disk, etc), some of which can be addressed differently (*e.g.*, through files).
- Some interfaces to the outside world, via peripherics (screen, mouse, Ethernet, etc).

By itself, the CPU only moves bytes around in memory, and can perform operation on them. It understands the concept of **integers** and **floating point numbers** (IEEE-754 shenanigans), but that's about it.

Most functionalities of a computer (*e.g.*, files) are in fact available thanks to the **operating system**, which offers, *e.g.*, an unified interface to peripherics.

The **assembler** is a pretty simple (and CPU-dependent) language. For example, it does not understand the concept of strings (which explains why Fortran and C implements them differently). And, among other things, *advanced* concepts like *loops* are not directly available.

So… A meme will now ensue.

The **assembler** is a pretty simple (and CPU-dependent) language. For example, it does not understand the concept of strings (which explains why Fortran and C implements them differently). And, among other things, *advanced* concepts like *loops* are not directly available.

So… A meme will now ensue.

(Note: it is in fact **conditional jumps**)

A programming language enables you to express complex logic (*e.g.*, loops, conditionals) in a structured and readable manner. This code is then either translated into machine instructions by a **compiler** or executed directly by an **interpreter**. While interpreted languages tend to be slower, they offer advantages such as dynamic execution (`exec()`), reflexivity (`getattr()`), and the ability to modify itself at runtime.

Python is an **interpreted** language, but it is also *just-in-time compiled* into an intermediate bytecode representation (stored in `__pycache__`). Over time, this process optimizes execution by reducing redundant checks, making subsequent runs of the code faster.

The way to write the code is referred to as a **programming paradigm**, a relatively high-level way to conceptualize and structure the implementation of a computer program.[1] Among others, there are:

- **Imperative**, in which the code directly controls execution flow and state change. This includes the famous **procedural** (`x = a(); y = b(x);`) and **object-oriented** (`x = X(); x.b();`) approaches.
- **Declarative**, in which code declares properties of the desired result, but not how to compute it, it describes what computation should be performed. This include the (in)famous **functional** approach (`(b(a()))`), but also programs based on **logic and constraints** (`x: int; y: int; x+y < 3;`).
- But also: **concurrent**, **visual**, etc...

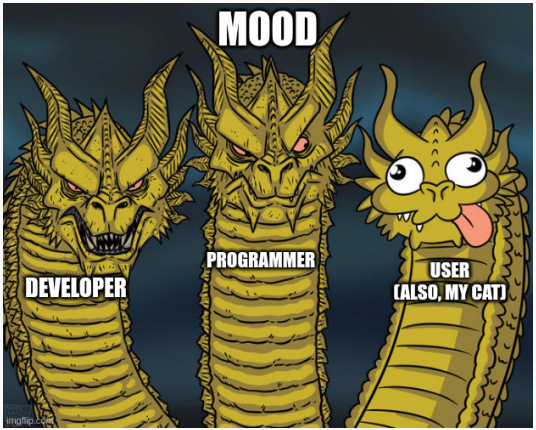
To a certain extent, all paradigm can be used in all languages. Python is generally approached as an OOP language.

[1] `https://en.wikipedia.org/wiki/Programming_paradigm`

Note on the “who”

In the rest of this presentation, I will (sometimes) distinguish three kinds of people:

1. The **developers**, who actually develop the code/library and eventually provide an API (*application programming interface*).
2. The **programmers**, who use the API provided by the code/library and develop on top of it.
3. The **users**, who use the program/executable, but do not program.



It’s all about abstraction

Abstractions

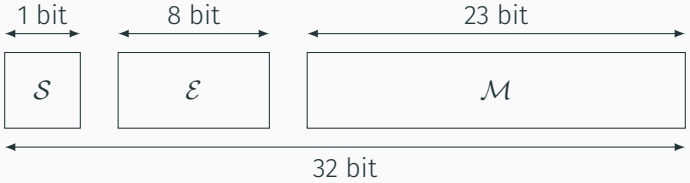
A well-designed program should be **modular**, with each module having clear and specific responsibilities. They can then work together . One of the best ways to achieve this is through **abstraction**, which means focusing on what matters while ignoring unnecessary details. There are several levels of abstraction:

1. **Abstraction by Parameterization**: Use parameters instead of hard-coded values to make code flexible and reusable.
2. **Abstraction by Procedures**: Structure your code using functions that call one another, rather than relying on *copy-paste-modify* patterns.
3. **Abstraction by Specification**: Describe **what** the code should do, not **how** it does it. This allows you to change the implementation without affecting other parts of the program. (More on this later.)
4. **Data Abstraction**: Instead of using raw data, encapsulate it in **abstract data types** (e.g., objects). Type hierarchies using **inheritance**, with child classes inheriting behaviors from parent classes. (More on this later.)
5. And others: syntaxic sugar, etc.

Example(s):

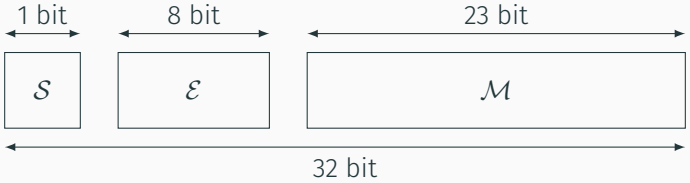
```
# without any abstraction
print('perimeter of circle is', 2 * 3.141592 * 3.5)
print('area is', 3.141592 * 3.5 ** 2)
# abstraction by parameterization
from math import pi as PI

r = 3.5
print('perimeter of circle is', 2 * PI * r)
print('area is', PI * r ** 2)
# abstraction by procedure
def info_circle(r: float) -> None:
  print('perimeter of circle is', 2 * PI * r)
  print('area is', PI * r ** 2)

info_circle(3.5)
info_circle(7.2)
```

An example of data abstraction:



$$x = (-1)^{\mathcal{S}} \times 2^{\mathcal{E}-2^7} \times \left(1 + \frac{\mathcal{M}}{2^{23}}\right).$$

An example of data abstraction:



$$x = (-1)^{\mathcal{S}} \times 2^{\mathcal{E}-2^7} \times \left(1 + \frac{\mathcal{M}}{2^{23}}\right).$$

And yet...

```
# let's play with floats:
x = 2.5e7  # S=0, E=151, M=4111392
y = x - 3.25e-2
```

This is a powerful abstraction!

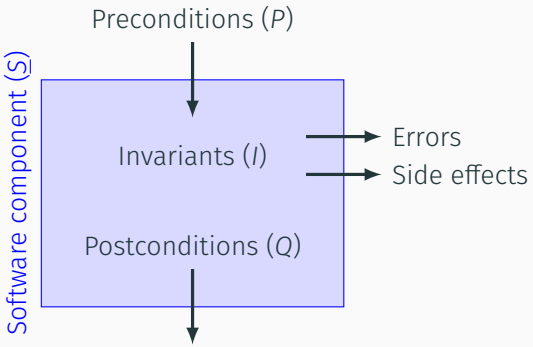
Abstraction by specification

Design by contract (i.e., abstraction by specification)

The “design by contract” approach prescribes that developers should define formal(?), precise(?) and verifiable(?) interface **specifications** for software components (functions, classes, modules, programs, etc).

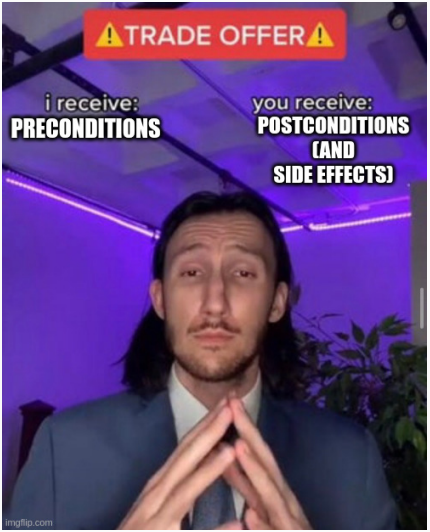
The “contract” is the following: $\{P \wedge I\}\ \underline{S}\ \{Q \wedge I\}$ ($\wedge$ side effects). Generally, $\neg(P \wedge I) \implies$ errors, but it is not necessarily true in all cases (and it does not have to be).

Notice that this does not say what is $\underline{S}$. This is thus mostly useful for the programmer (and the user).



\+ Eventual performances guarantees.

- The **preconditions** are predicates that must be true before the execution of the component (it generally boils down to the type of the inputs and their respective domains). It must be guaranteed by **the caller**. If a precondition is violated, the effect becomes undefined.
- The **postconditions** are predicates that must be true after execution (if the preconditions are true), which is guaranteed by the **callee**.
- The **invariants** are predicates that must be true before and after execution. This is guaranteed by the caller **and** the callee (but mostly the later).



Example: $P \equiv x \in \mathbb{R}$ (not necessarily true in python!), $Q \equiv \mathcal{R} = |x|$, $I \equiv$ yes, $\underline{S} \equiv$

```
def _abs(x: float) -> float:
  return -x if x < 0 else x  # note: that's called syntaxic sugar
```

Here, $sp(\underline{S}, P \wedge I) \implies Q \wedge I$, so this program is valid. Indeed,

$$\begin{aligned} Q' &= sp(\texttt{return -x if x < 0 else x}, x \in \mathbb{R}) \\ &= (\mathcal{R} = -x \wedge x < 0 \wedge x \in \mathbb{R}) \vee (\mathcal{R} = x \wedge x \geq 0 \wedge x \in \mathbb{R}) \\ &= [(\mathcal{R} = -x \wedge x < 0) \vee (\mathcal{R} = x \wedge x \geq 0)] \wedge x \in \mathbb{R} \end{aligned}$$

and $Q' \implies Q \wedge I$. Note that `_abs("test")` is indeed undefined.

This is how you should ensure that your programs are valid, except nobody does that in practice (except in security contexts, like planes, trains, or in space). Note that one of the implicit assumption is that the interpreter works correctly (so you also need to "prove" the interpreter if you want to be 100% correct).

- **Errors:** we will see below how to play with errors, but if one sticks to *design by contract* alone, this is not required, since the preconditions **must always** be fulfilled. In practice, it is generally not the case (*never trust user input*!), and having undefined behaviors may lead to unexpected bugs (since the execution can continue) or even security issues. Thus, one can abide to **defensive programming** instead, which consists in weakening the precondition (or delete it), and either report failure (throwing exceptions) or running in degraded mode.[2]
- This leads to **side-effects**, which is the fact that the code have an observable effect other than reading its input and giving an output. Other example of side-effects include: performing I/O, modifying a non-local variable (*e.g.*, matplotlib's `pyplot` interface, which is a state machine), or mutate an argument (*e.g.*, sorting a list in-place).[3] **All** such side-effects should be **properly** documented.

[2] Purely failing would be a bit annoying in, *e.g.*, a plane ;)

[3] BLAS and LAPACK are customary of such practices.

In practice, be nice to your programmers/users (… and to future you, for that matter) that your **specifications** contains the following:

- Document your inputs, it is better than nothing. This include their types (in some language, it is in the definitions) and their **domain** (*e.g.*, for `sqrt(x)`, `x >= 0`).
- Document the **effect** of the code rather than what it does (which does not prevent you to have another document which details the algorithms).
- If possible, document all **side effects**, in particular if you mutate an argument.

Since you will probably never properly prove your pre/postconditions, you don't have to use a mathematical language to express everything.[4] Even if any spoken language is ambiguous, a nice textual description is better than nothing.

The goal is therefore to provide a "framework" for things you should document, rather than a set of rules.

[4] personally, I write my pre/post-conditions using the programming language itself, for example `x >= 0 and 0 <= y < 3 and z in ['a', 'b', 'c']`.

Caution:

Programming by contract assume that **everyone** write, read, understand, and follow the specifications.

That's rarely the case in amateur projects (including some of mine, by the way).

Documentation(s?)

- The documentation is therefore the place where you indicate your pre/post-conditions, side effects, etc. Note that these concepts applies to any component you can think of (even a CLI program has inputs and outputs), although with different names. Again, “framework” rather than set of rules.
- Be nice to your programmers/users: details the installation procedure as much as possible (detail the dependencies if any), provide examples,[5] Detail the contribution procedure, etc. We all have in mind good and bad documentations... Let’s not be an example of the latter ;)
- There is no convention for the form of your documentation (it can be done with any tool you want), but it is generally better to diminish the amount of manual work, which means that documentation **auto-generation** is a good idea.

[5] ... But an example alone is not a good documentation, thought.

The idea of auto-generation is to write a part of your documentation directly in the code as comments, then to use a tool that read such comments and generate a nice document (generally a set of HTML pages) as output.

The tool of the trade here is generally Doxygen (see `https://www.doxygen.nl/`), which was first created for C++:

```cpp
class Dog {
  public:
  /**
  * \brief Creates a new Dog.
  *
  * Creates a new Dog named `_name`.
  *
  * \param _name The dog's name.
  */
  Dog(const char* _name);
};
```

… But works for many languages (including Python and Fortran).

But in Python, nobody agrees yet.[6] There are three concurrent documentation formats (ReST, numpy, and Google) and a plethora of tools to interpret them, among which:

- Doxygen, which has its own format ;
- autodoc (used with Sphinx), which requires to write in ReST (more complete than markdown, but more complex) ;
- mkdocstring (used with mkdocs), which uses markdown ;
- pdoc3, which uses markdown.

All four generates an HTML output (which is generally what you want). They supports LaTeX formula by various means.

[6] Sphinx seems to win for major projects, thought.

DEMO TIME!

- Sphinx
- `python -m doctest -v neighbour_search/*.py`

Error handling (*exceptions*)

As mentioned above, there are two possible reasons for reporting an error:

1. When the precondition is not fulfilled. Again, you should normally not check that, and thus, not report an error for that. However... This is still useful for the developer during implementation. Thus, **if checking (a part of) the precondition is cheap**,[7] you can use `assert`, which is a function/statement available in many languages that is only compiled/interpreted in "debug" mode.[8]
2. When you want to avoid undefined behavior by weakening the preconditions and issuing errors instead.

[7] $\mathcal{O}(1)$, no more.
[8] calling `python -O -c "assert false"` results in no error.

And there are two ways to report an error. One is to use the return value or mutate a global variable (or, on languages that allows it, mutate an input designated for that, like `ierror`):

```python
def normalize(x: np.ndarray) -> int:
  """
  pre: x is a vector
  post: if return value is 0, x is normalized, if return value is 1, it is not.
  """
  norm = np.linalg.norm(x)
  if norm == 0:
    return 1
  x /= norm
  return 0
```

Although totally fine, this is a bit long. Also, the caller can ignore the return value,[9] which would in practice create an undefined behavior.

[9] Many bugs in old AND modern programs are due to non-checked return values when allocating memory.

This is why many “modern” programming languages propose a mechanism called **exceptions**. It works like this:

- When an issue should be issued, the code **throw** an exception (which can be an integer, a string, or something more complex like and object which allows to provide context).
- The execution propagate up in the call stack, to a point where the error can be handled. If there is none, the execution terminates badly.

This force the caller to handle the exception. It also effectively separate the code that detect the error and the one that handles it.

DEMO TIME!

The take home message here is:

- `assert` for the developers,
- `throw` (+ documentation) for the programmers, and
- a nice error message (eventually in `stderr`) for the users. The stack trace frighten them.

And, by the way...

The take home message here is:

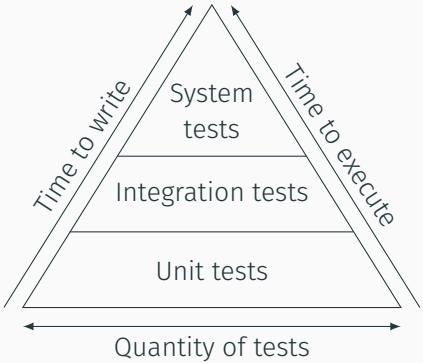
- `assert` for the developers,
- `throw` (+ documentation) for the programmers, and
- a nice error message (eventually in `stderr`) for the users. The stack trace frighten them.

And, by the way...

Testing

Per Wikipedia, “software testing is the act of checking whether software satisfies expectations”.

It is **very** important, since it provides ways to check whether your code do what it is supposed to do without proving your code… Which does not guarantee that your code is correct, just that it seems to work under certain conditions, which is already something ;)



(other kind of tests exists)

- **Unit tests**: they test small parts of your code (*i.e.*, a function or a class) and check its behavior. They are quick to run, and are written with the actual implementation in mind (white box testing) and maximize **coverage** (*i.e.*, most if not all execution paths are checked). In some case, they go beyond checking that the pre/postconditions are true.[10] They generally use mock data.
- **Integration tests**: checks that the functions, classes, modules, etc of your project works together. They are higher level, and are not written with implementation in mind, only the specifications (black box testing). They generally use mock data or very simple real ones (think water in HF/STO-3G to check Mulliken charges).
- **System tests** (also called functional tests): high-level tests (to check that everything works together) with real data from real life. They generally check that the front-end program/library meets its requirements (think real medium-sized molecule, with a realistic method and basis set). They are generally manually launched!

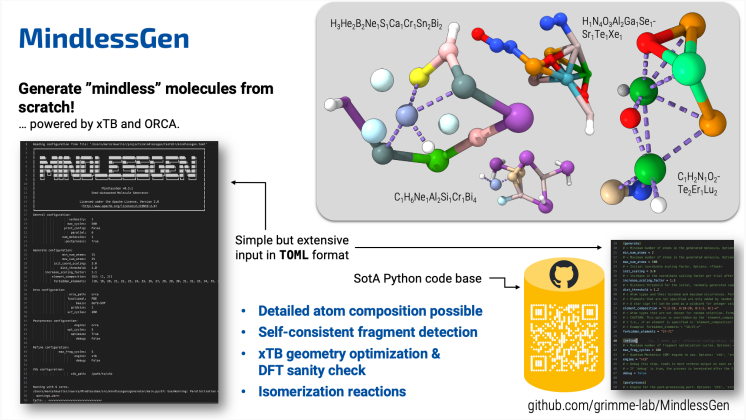
All these constitute your **tests suite**.

[10] The pre/postconditions can be weaker than the actual implementation.

Ideally, tests should:

- Be written as soon as possible. In fact, if you already have the specifications, you can start by writing (unit) tests that check those specifications before even writing the implementation. This is **test-driven development**.
- Run often. A good practice is to have a **continuous integration (CI) pipeline**, which generally boils down to a script that runs all your unit and integration tests, automatically launched at every occasion (commit, PR, etc). This helps you to prevent **regressions**.
- Fail quickly. If you have to wait 20 minutes to know if your implementation fails, that's painful. This is why most tools allow you to run only one part of your test suite locally (and this is why **mock data** are useful).
- Be smart: no need to test EVERY input, only a few well-chosen ones ;)
- Test as much as possible, but not too much: no need to test that a dependency that you use works as intended, that's their job not yours!

Speaking of mock data… Yet another *“grimmerie”*:



https://github.com/grimme-lab/MindlessGen/

How to write tests:

- Follow the following structure:
  1. Setup: prepare any data you need for your test ;
  2. Stimuli: run the piece of code you want to test ;
  3. Verify: check that the output matches what you expect ;
  4. Teardown: clean up your mess, if any.

  In general, the first and last steps are common to many tests (*e.g.*, the starting geometry of water is the same for an energy or optimization calculation).
- Separate your tests from your code: you may want to ship your library/program without its tests for actual use.
- Use a proper test framework! In python, this means either `unittest` or `pytest`, which both provide ways to call your test suite (or some parts of it) smoothly.

DEMO TIME!

- `pytest` basics (create and run)
- `pytest.approx()` and `numpy.allclose()`
- fixtures

Data abstraction

Data structures

The idea behind common data structures (list, arrays, dictionary, sets, etc) is to provide efficient ways to store and retrieve data depending on the problem you want to solve.

To measure the efficiency, one measures the **computational complexity** of different tasks, defined as:

$$T(n) = \mathcal{O}(g(n)) \text{ as } n \to \infty \implies \exists M, n_0 \in \mathbb{N}^+ : \forall n \geq n_0 : |T(n)| \leq M|g(n)|,$$

where $T(n)$ is the function that measure the execution time of a given piece of code (generally a function) given an input of size n. In a sense, the big-$\mathcal{O}$ refers to the asymptotic behavior of $T(n)$ (and is thus also found when speaking about finite series).

$g(n)$ is generally taken as simple as possible, such as 1, $\log_2 n$ (generally written $\log n$), n, etc.

Example:

```python
def find1(a: list[int], v: int) -> int:
  """
  Get the position of `v` in `a`.

  pre: `a` is a list of integers, `v` is an integer.
  post: if it exists in `a`, the index of `v` is returned if in `a`. A negative
  ↪  value is returned otherwise.
  """
  for i in range(len(a)):
    if a[i] == v:
      return i
  return -1   # note: postcondition is less strict than actual implementation
```

In the worst case scenario (`v` is not in `a`), the loop is executed n times, where n is the size of `a`. This algorithm is thus $\mathcal{O}(n)$.

```python
def find2(a: list[int], v: int) -> int:
  """
  Get the position of `v` in `a`.

  pre: `a` is a **sorted** list of integers, `v` is an integer.
  post: if it exists in `a`, the index of `v` is returned if in `a`. A negative
  ↪  value is returned otherwise.
  """
  b, e = 0, len(a) - 1
  while e >= b:  # note: T(n) = 1 + T(n/2)
    p = (e + b) // 2  # note: integer division
    if a[p] == v:
      return p
    elif a[p] > v:
      e = p - 1    # then it's left
    else:
      b = p + 1   # then it's right
  return -1
```

In the worst case scenario, the loop is executed $\log_2 n$ times, so $\mathcal{O}(\log n)$.

In a nutshell:

Data structure	Average time complexity				Worst time complexity			
	Access	Search	Insert	Delete	Access	Search	Insert	Delete
Array	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
List	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
BS-tree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
(R)B-tree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Hash Table	—	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	—	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Table 1: Data structure operation costs, from `https://www.bigocheatsheet.com/`.

Notes:

- In practice, nobody care how its implemented.
- All these structure have $\mathcal{O}(n)$ space complexity.
- Stack, Queue, List, etc are generally implemented as singly/doubled linked lists.
- Hash tables (dictionary) are generally a combination of an array (for which the hash function give the key) and some sort of list or tree to solve collisions.
- The `set` (and `frozenset`?) is implemented as a hash table.
- See below for BS (binary search) tree.

Oriented object programming (OOP)

The goal of OOP is to define new **abstract data types** (ADT), *i.e.*, separate the concrete representation of the data and their (abstract) signification. For example, a *molecule* (note that you should not define members like this):

```python
class Atom:
  symbol: str
  coordinates:
  ↪ list[float]

class Molecule:
  atoms: list[Atom]
```

```python
class Molecule:
  symbols: list[str]
  positions:
  ↪ np.ndarray
```

```python
class Atom:
  symbol: str
  coordinates:
  ↪ list[float]

class Molecule:
  atoms:
  ↪ SearchTree[Atom]
```

In all three cases, although the **internal representation** (irep) of the molecule change, its behavior should not! And the programmer should not knwow (or care) about what is inside.

So, what do you need to do OOP?

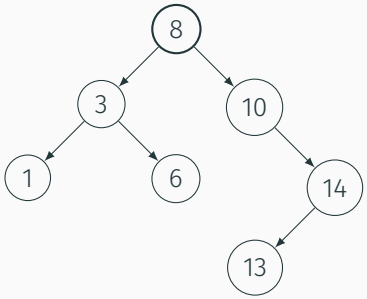
- The **need** for an ADT (it is more frequent than you think!).
- An (abstract) **specification**: what your ADT is suppose to do? (again, focus on "what", not "how"). This may include an **ADT invariant** (AI), something that should be true during the lifetime of an instance of the ADT.
- **The specification the methods**: these methods provides ways to interact with your ADT: normally, the programmer should never directly interact with your irep.[11] Furthemore, no method should break the AI!
- **An irep**: it should be chosen so that it allow you to implements all specified methods. A useful tool in there is the definition of a **representation invariant** (RI), which translates the ADT invariant in concrete code.

In python (and in other languages), an ADT is implemented via a `class`. Then, in the rest of the code, you **instanciate** your ADT and interact with these instances with the methods that where defined in the `class`.

[11] This is called **encapsulation**. Python is rather permissive here.

(textbook) example: a **binary search (BS) tree**:

- The **need**: yes. Data are automatically sorted as a result!
- An (abstract) **specification**: a binary search tree is a tree that allow binary search for fast lookup, addition, and removal of data items (here, integers). Each node of the tree has (at most) two children, the *left* and the *right*. The AI is the following: "each node is greater than all the ones in its left subtree and less than the ones in its right subtree".
- **The specification the methods**: one needs at least `insert(k: int)`, `delete(k: int)`, and `search(k: int)`.
- **An irep**: linked lists would do, but funny implementations exists with arrays.



Questions: how to perform:

1. `search(6)`, or
2. `insert(12)`, or
3. `delete(8)`,

in order to keep the AI valid? And in the second case, what is the complexity?

Useful methods:

- **Creators** (including, but not limited to, *constructors*): allow to create instances of your ADT out of nowhere.
- **Producers**: create new instance from existing one(s). Particularly useful for **immutable** ADT, but you need to take care of the copy of your irep.[12]
- **Modifiers** (also called *setters*): modifies the instance directly.
- **Observers** (also called *getters*): provide access to properties of the instance.

A similar nomenclature is CRUD (create, read, update, delete[13]). Python also have *magic* methods,[14] `__func__()` which, if you define them, allow to use syntaxic sugar with your ADT (*e.g.*, `x = A(); y = A(); z = x + y`). One of them is the *constructor*.

[12] Shallow versus deep copy.
[13] There is no deleter in Python.
[14] See `https://docs.python.org/3/reference/datamodel.html#special-method-names`

Hierarchy of type (i.e., inheritance)

The **specialization** of an ADT is possible via a mechanism called inheritance.[15] The rule being, theoretically, that "*if S is a subtype of T, S can be used anywhere where T is used*".[16] Python rather uses the concept of *duck typing*: "an object is of a given type if it has (all) the methods of that type".[17]

In practice, given a parent class, a children class inherit all its parents methods, as well as its invariants (which it should not break) and irep (which it should normally not access directly). It can however re-implement (*override*) some of the parent methods (but should keep their signature!) as well as add new ones.

It is also possible in certain language to define **interfaces**, that only provide specifications, not actual implementation. This is useful in the case of a base ADT that can be extended in many different ways.

[15] Other mechanism, like composition/traits, are possible.
[16] This is known as the *Liskov substitution principle* and this is… Debated.
[17] `https://en.wikipedia.org/wiki/Duck_typing`

DEMO TIME!

A few Python’s specifics

Project organization

The following organization is recommended:

- `package1/`
  - `__init__.py`: this file should be present if you want Python to recognize the directory as a package.
  - `x.py`: use `from package1.x import y` anywhere in your project to import things from this file.
  - `...`
- `package2/`: a project can contain more than one package.
- `tests/`: tests files (can be also in each package if you prefer)
- `docs/`: documentation files
- `README.md`: The starting point for any info
- `LICENSE`: How do you want your project to be reused?
- `pyproject.toml`: Tell Python about your project! Thanks to that file, it knows what to install and how.[18]

[18] See `https://packaging.python.org/en/latest/guides/writing-pyproject-toml/`, regularly.

Use a **virtual environment**: it is "used to contain a specific Python interpreter and software libraries and binaries which are needed to support a project (library or application). These are by default isolated from software in other virtual environments and Python interpreters and libraries installed in the operating system."[19]

The workflow is the following:

```
# create a new virtualenv
python -m venv venv
# use venv
source venv/bin/activate
pip install ...
python ...
# get out of venv
deactivate
```

[19] https://docs.python.org/3/library/venv.html

DEMO TIME!

(`pyproject.toml` and virtualenv, don't forget `-e.[dev]`)

And don't forget: **use git**, so...

- `.gitignore`: you can use `http://gitignore.io/` to create one.
- `.github/`: automatically launch tests, do lints, build docs, etc (only on Github).

Writing “beautiful” Python

Do you like writing beautiful code? Afraid of missing something? Ever dreamed of getting screamed at by Python? Say no more!

- Don’t reinvent the wheel: the standard library of Python is packed with good stuffs, try them! Also, the `scipy` stack is awesome for us.
- Follow the “zen of Python”[20] and the other philosophical principles/maxims of the Python community.
- use `flake8`: checks that your code follows the Python style guidelines. These are a set of rather strict rules, but you can modulate them by using a `.flake8` file.[21]
- Use `mypy`: checks that your type annotation make sense. It might seems useless or trivial, but don’t forget that those pesky `None` can get in the way (generating unexpected bugs).

[20] `https://peps.python.org/pep-0020/`
[21] `https://flake8.pycqa.org/en/latest/`

Conclusion(s?)

Programming **is** fun: don’t forget to have fun ;)

(but maybe think a bit before you implement something)

(although a project should always be done twice)