Introduction

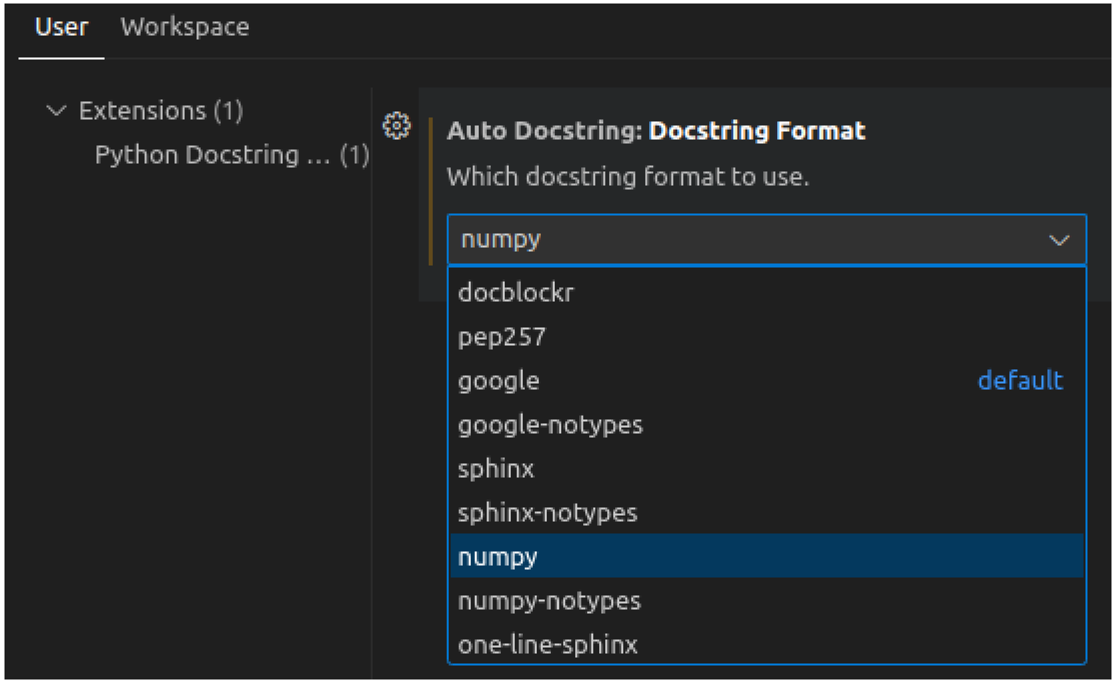
Documenting a code consists in writing ***docstrings***. *Docstrings* are special strings attached to a code section that a have special meaning for the interpreter. They are what is displayed using the `help` function.

In this part, the `src` package introduced previsouly is documented using VSCode (see file `module_2_A_1.py`).

Setting up the needed tools

A plugin is needed to write proper docstrings: **autoDocstring** (Nils Werner).

A docstring contains some important information (attributes, types, explanations) that must be formatted in a consistent way. Several formatting mode exist, but a common one is the `numpy` formatting mode.



Auto Docstring: Docstring Format

Which docstring format to use.

numpy

- docblockr
- pep257
- google (default)
- google-notypes
- sphinx
- sphinx-notypes
- numpy
- numpy-notypes
- one-line-sphinx

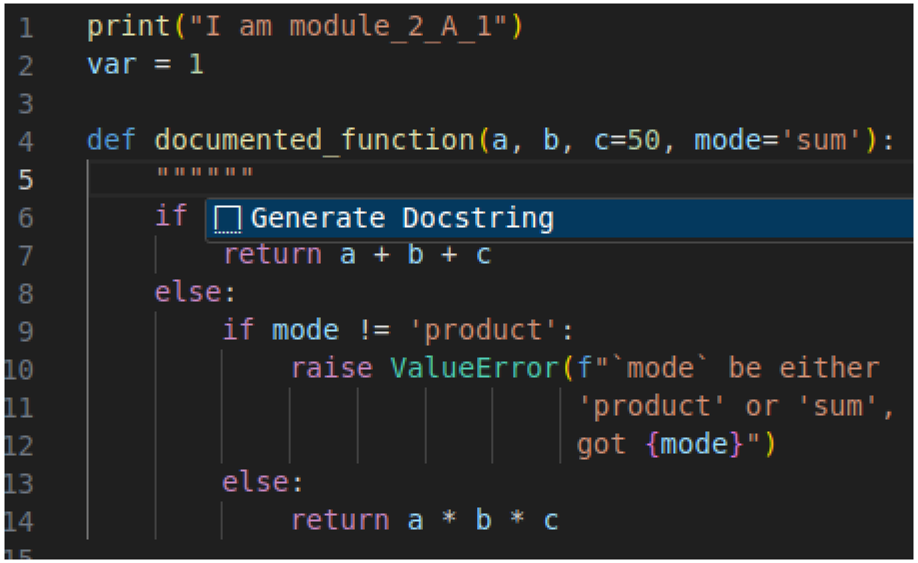
This format follows these rules. A quick overview is shown here after:

In [1]:
```
def abc(a: int, c = [1,2]):
    """_summary_

    Parameters
    ----------
    a : int
        _description_
    c : list, optional
        _description_, by default [1,2]

    Returns
    -------
    _type_
        _description_

    Raises
    ------
    AssertionError
        _description_
    """
    if a > 10:
        raise AssertionError("a is more than 10")

    return c
```

Create a docstring

One can create docstrings for **modules, functions, classes and methods**:

1. Place the carret immediately after the definition line (ex: `def` or `class`)
2. write `"""`
3. press `enter`



```python
print("I am module_2_A_1")
var = 1

def documented_function(a, b, c=50, mode='sum'):
    """"""
    if
        return a + b + c
    else:
        if mode != 'product':
            raise ValueError(f"`mode` be either
                             'product' or 'sum',
                             got {mode}")
        else:
            return a * b * c
```

Add some content to a docstring

Key idea

Prioritary information is given first:

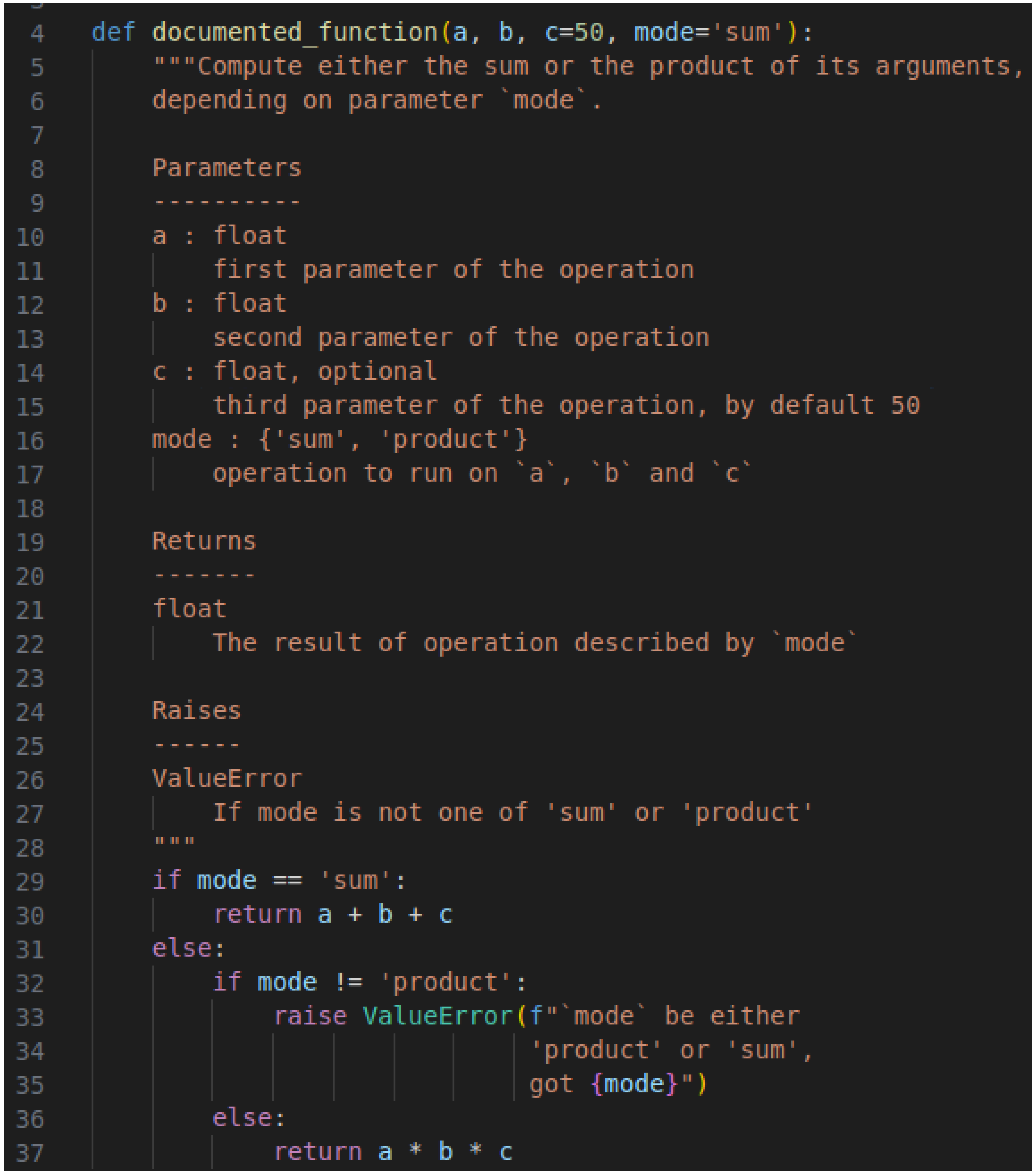
1. Purpose of the function
2. Input parameters
3. Returned parameters

Some reminders:

- functionalities of the code are first described using a software point of view.

  Then, a scientific explanation is added if needed.

- imperative mood must be used



```python
def documented_function(a, b, c=50, mode='sum'):
    """Compute either the sum or the product of its arguments,
    depending on parameter `mode`.

    Parameters
    ----------
    a : float
        first parameter of the operation
    b : float
        second parameter of the operation
    c : float, optional
        third parameter of the operation, by default 50
    mode : {'sum', 'product'}
        operation to run on `a`, `b` and `c`

    Returns
    -------
    float
        The result of operation described by `mode`

    Raises
    ------
    ValueError
        If mode is not one of 'sum' or 'product'
    """
    if mode == 'sum':
        return a + b + c
    else:
        if mode != 'product':
            raise ValueError(f"`mode` be either
                             'product' or 'sum',
                             got {mode}")
        else:
            return a * b * c
```

Notes: all references to a software element (variable, module, function and classes) must be quoted with **backticks**: `` ` `` (`Alt Gr + 7` on a french keyboard).

An iterative process

It is very common to discover weaknesses in the code while writing docstrings:

- possibility of erroneous scientific results
- unconsistent code from one component to another
- instability risk
- ...

For these reasons, the preferred way of writing documentation is first documenting all the components without any detail, and then go further when additional information is needed.