Computer Systems Modeling and Verification
(USEEN1)

Functions, procedures, and effects*

Tristan Crolard

Department of Computer Science
CEDRIC lab / SYS team

tristan.crolard@cnam.fr

cedric.cnam.fr/sys/crolard

*. These slides are adapted from *Python for Computational Science* (2024)

Effects

When a function has some unexpected effects, we talk about side effects. Example:

```
>>> def sum(xs: list[int]) -> int:
        s = 0
        for i in range(len(xs)):
            s = s + xs.pop()
        return s
>>> xs = [10, 20, 30]
>>> print("xs␣=", xs)
xs = [10, 20, 30]
>>> print("sum(xs)␣=", sum(xs))
sum(xs) = 60
>>> print("xs␣=", xs)
xs = []
```

Effect-free functions

Better ways to compute the sum of a list `xs` (or sequence in general):

- use indices to iterate over list

```
>>> def sum(xs: list[int]) -> int:
        s: int = 0
        for i in range(len(xs)):
            s = s + xs[i]
        return s
```

- or better: iterate over the elements directly

```
>>> def sum(xs: list[int]) -> int:
        s: int = 0
        for elem in xs:
            s = s + elem
        return s
```

Mutable and immutable sequences

Effects can be made explicit using the type hints: collection interfaces (called Abstract Base Classes[1] in Python) may be used to provide types to arguments instead of concrete implementations.

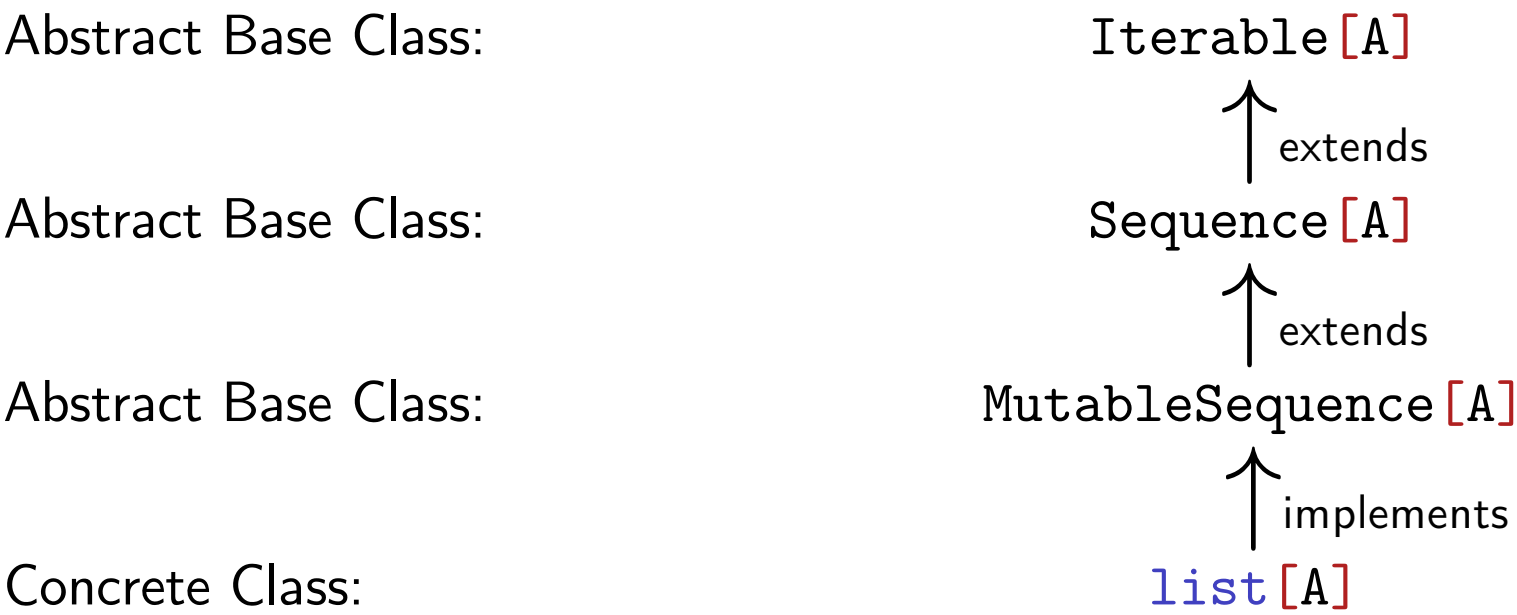
For instance, instead of `list[A]`, use:

- `Sequence[A]` whenever the argument is not modified by the function
- `MutableSequence[A]` otherwise.

Moreover, the type checker will ensure that a `Sequence[A]` is not modified.

1. https://docs.python.org/3/library/collections.abc.html#collections-abstract-base-classes

Sequences types: `list`



Note. You cannot create an instance of an Abstract Base Class, but they are useful for typing variables and parameters.

Sequences types: `str`

Abstract Base Class: `Iterable[str]`

Abstract Base Class: `Sequence[str]`

Concrete Class: `str`

Note. It should really be `Iterable[char]` and `Sequence[char]`, but there is no `char` type in Python.

Functions, procedures and effects

- A function that exits through the `return` keyword, will return the object given after `return`.
- A function that does not use the `return` keyword, implicitly returns the special object `None`. Such a function is also called a procedure.
- Procedure should either modify their arguments or perform some other effect (such as printing).
- Functions should not modify their arguments but they should return a value. Such regular functions are sometimes called effect-free.
- Functions that both modify their arguments and return a value are called effectful.

Note. Calling functions from the prompt can cause some confusion here: if the function returns a value, it will also be printed (even if the function is effect-free).

Function and procedure – examples

```
>>> from collections.abc import Sequence, MutableSequence
```

- **Function**

```
>>> def count_zeros(xs: Sequence[int]) -> int:
        count: int = 0
        for elem in xs:
            if elem == 0:
                count = count + 1
        return count
```

- **Procedure**

```
>>> def remove_zeros(xs: MutableSequence[int]) -> None:
        i: int = 0
        while i < len(xs):
            if xs[i] == 0:
                xs.pop(i)
            else:
                i = i + 1
```

Effectful function – example

Sometimes, you might want to do both at the same time:

```
>>> def count_and_remove_zeros(xs: MutableSequence[int]) -> int:
        i: int = 0
        count: int = 0
        while i < len(xs):
            if xs[i] == 0:
                xs.pop(i)
                count = count + 1
            else:
                i = i + 1
        return count
```

Note. This is usually not a good idea: you should avoid merging functions and procedure into a single code like that (you should first check if this optimization is required).

Testing functions and procedures – example

- **Function**

```
>>> def test_count_zeros() -> None:
        expected = 2
        actual = count_zeros([1, 0, 7, 0, 9])
        assert expected == actual
```

- **Procedure**

```
>>> def test_remove_zeros() -> None:
        xs = [1, 0, 7, 0, 9]
        remove_zeros(xs)
        assert xs == [1, 7, 9]
```

Testing effectful functions – example

```
>>> def test_count_and_remove_zeros() -> None:
        expected = 2
        xs = [1, 0, 7, 0, 9]
        actual = count_and_remove_zeros(xs)
        assert xs == [1, 7, 9]
        assert expected == actual
```

Note. Both effect-free functions and procedures are special cases of effectful functions, but you should restrict yourselves to those as far as possible: unrestricted effectful functions are more difficult to write and they are also more difficult to test.

Default argument values

- **Motivation**
  - suppose we need to compute the area of rectangles and we know the side lengths `a` and `b`.
  - Most of the time, `b=1` but sometimes `b` can take other values.
- **Solution 1**:

```
>>> def area(a: int, b: int) -> int:
        return a * b
```

```
>>> print("The␣area␣is", area(3, 1))
```

```
The area is 3
```

```
>>> print("The␣area␣is", area(4, 1))
```

```
The area is 4
```

- We can make the function more user friendly by providing a default value for `b`. We then only have to specify `b` if it is different from this default value.

- **Solution 2** (with a default value for argument b):

```
>>> def area(a: int, b: int = 1) -> int:
        return a * b
>>> print("The␣area␣is", area(3))
The area is 3
>>> print("The␣area␣is", area(4))
The area is 4
>>> print("The␣area␣is", area(4, 2))
The area is 8
```

- If a default value is defined, then this parameter (here b) is optional when the function is called.
- Default parameters have to be at the end of the argument list in the function definition.

Default argument values – examples

You have met default arguments in use before, for example:

- the `print` function uses the following default values:
  `end = '\n'`
  `sep = '␣'`

- the `list.pop` method uses the following default value:
  `index = -1`

Keyword argument values

- We can call functions with a "keyword" and a value (the "keyword" is the name of the formal parameter in the function definition).
- Here is an example:

```
>>> def f(a: int, b: int, c: int) -> None:
        print("a=", a, ", b=", b, ", c=", c, sep = "")
```

```
>>> f(1, 2, 3)
```

```
a=1, b=2, c=3
```

```
>>> f(a=3, b=1, c=2)
```

```
a=3, b=1, c=2
```

```
>>> f(1, b=3, c=2)
```

```
a=1, b=3, c=2
```

- keyword arguments are often used in combination with default values.