Fonctions d'ordre supérieur*

Tristan Crolard

Laboratoire CEDRIC
Equipe « Systèmes Sûrs »

tristan.crolard@cnam.fr

cedric.cnam.fr/sys/crolard

*. Ces supports sont adaptés de *Python for Computational Science* (2024)

Higher Order Functions

Motivational exercise: function tables

- Write a function `print_x2_table` that prints a table of values of $f(x) = x^2$ for $x = 0, 1, 2, \ldots, 5$ i.e.

```
0 0
1 1
2 4
3 9
4 16
5 25
```

- Then do the same for $f(x) = x^3$
- Then do the same for $f(x) = 5x + 3$

  ...

Higher Order Functions – 2

Solution:

```
>>> def print_x2_table() -> None:
        for x in range(6):
            print(x, x ** 2)
```

```
>>> print_x2_table()
```

```
0 0
1 1
2 4
3 9
4 16
5 25
```

Higher Order Functions – 3

Idea: Pass function `f` to tabulating function.

Example:

```
>>> from collections.abc import Callable
>>> def print_f_table(f: Callable[[int], int]) -> None:
        for x in range(6):
            print(x, f(x))
>>> def square(x: int) -> int:
        return x ** 2
>>> print_f_table(square)
0 0
1 1
2 4
3 9
4 16
5 25
```

```
>>> def cubic(x: int) -> int:
        return x ** 3
```

```
>>> print_f_table(cubic)
```

```
0 0
1 1
2 8
3 27
4 64
5 125
```

```
>>> def g(x: int) -> int:
        return 5 * x + 3
```

```
>>> print_f_table(g)
```

```
0 3
1 8
2 13
3 18
4 23
5 28
```

Returning “function objects”

We have seen that we can pass function objects as arguments to a function. Now we look at functions that return function objects.

Example

```
>>> def make_add42() -> Callable[[int], int]:
        def add42(x: int) -> int:
            return x + 42
        return add42
```

```
>>> add42 = make_add42()
```

```
>>> print(add42(2))
```

```
44
```

Closures

A "function object" is often called a closure because some variable can be captured when the function is defined and this variable has to be bundled (enclosed) together with the function.

Example

```
>>> def make_adder(y: int) -> Callable[[int], int]:
        def adder(x: int) -> int:
            return x + y
        return adder
>>> add42 = make_adder(42)
>>> add42(2)
44
```

Anonymous function – lambda expressions

- lambda expressions: anonymous function
- The following syntax defines a function object.
  `lambda parameters: expression`
- Useful to define a small helper function that is only needed once

```
>>> lambda a: a
<function <lambda> at 0x1005fd6c0>
>>> lambda a: 2 * a
<function <lambda> at 0x1005fd620>
>>> (lambda a: 2 * a)(10)
20
```

```
>>> lambda a: a
<function <lambda> at 0x1005fd6c0>
>>> lambda a: 2 * a
<function <lambda> at 0x1005fd620>
```

```
>>> (lambda a: 2 * a)(10)
20
>>> (lambda a: 2 * a)(20)
40
>>> (lambda x, y: x + y)(10, 20)
30
>>> type(lambda x, y: x + y)
<class 'function'>
```

Some predefined higher order functions

https://docs.python.org/3/library/functions.html#built-in-functions

Rough summary (without type hints)

- `map(function, iterable) -> iterable`
  apply `function` to all elements in `iterable`
- `filter(function, iterable) -> iterable`
  return items of `iterable` for which `function(item)` is true.
- `reduce(function, iterable, initial) -> value`
  apply `function` from left to right to reduce iterable to a single value.

Note. Remember that sequences are iterables.

Map

```
map[A, B](f: Callable[[A], B], it: Iterable[A]) -> Iterable[B]
```

Example

```
>>> def f(x: int) -> int:
        return x ** 2
>>> list(map(f, [0, 1, 2, 3, 4]))  # convert list to iterable
```

```
[0, 1, 4, 9, 16]
```

Using a lambda expression:

```
>>> list(map(lambda x: x ** 2, [0, 1, 2, 3, 4]))
```

```
[0, 1, 4, 9, 16]
```

Using list comprehension:

```
>>> [x ** 2 for x in [0, 1, 2, 3, 4]]
```

```
[0, 1, 4, 9, 16]
```

Map – implementation

Using a for loop

```
>>> from collections.abc import Callable, Iterable
```

```
>>> def map[A,B](f: Callable[[A], B], it: Iterable[A]) -> Iterable[B]:
        l: list[B] = []
        for element in it:
            l.append(f(element))
        return l
```

Using list comprehension

```
>>> def map[A,B](f: Callable[[A], B], it: Iterable[A]) -> Iterable[B]:
        return [f(element) for element in it]
```

Filter

```
filter[A](f: Callable[[A], bool], it: Iterable[A]) -> Iterable[A]
```

Example

```
>>> def is_positive(n: int) -> bool: # check if n is positive
        return n > 0
>>> list(filter(is_positive, [-3, -2, -1, 0, 1, 2, 3]))
```

```
[1, 2, 3]
```

Using a lambda expression:

```
>>> list(filter(lambda n: n > 0, [-3, -2, -1, 0, 1, 2, 3, 4]))
```

```
[1, 2, 3, 4]
```

List comprehension equivalent:

```
>>> [n for n in [-3, -2, -1, 0, 1, 2, 3, 4] if n > 0]
```

```
[1, 2, 3, 4]
```

Filter – implementation

Using a for loop

```
>>> from collections.abc import Callable, Iterable
```

```
>>> def filter[A](f:Callable[[A],bool],it:Iterable[A]) -> Iterable[A]:
        l: list[A] = []
        for element in it:
            if f(element):
                l.append(element)
        return l
```

Using list comprehension

```
>>> def filter[A](f:Callable[[A],bool],it:Iterable[A]) -> Iterable[A]:
        return [element for element in it if f(element)]
```

Reduce

```
reduce[A, B](f: Callable[[A, B],A], it: Iterable[B], init: A) -> A
```

Example

```
>>> from functools import reduce
```

```
>>> def f(x: int, y: int) -> int:
        print("Called␣with␣x␣=", x, "y␣=", y)
        return x + y
```

```
>>> reduce(f, [1, 3, 5, 7, 9], 10)
```

```
Called with x = 10 y = 1
Called with x = 11 y = 3
Called with x = 14 y = 5
Called with x = 19 y = 7
Called with x = 26 y = 9
35
```

Reduce – implementation

https://docs.python.org/3/library/functools.html#functools.reduce

```
>>> def reduce[A,B](f:Callable[[A,B],A], it:Iterable[B], init:A) -> A:
        value: A = init
        for element in it:
            value = f(value, element)
        return value
>>> reduce(f, [1, 3, 5, 7, 9], 100)
Called with x = 100 y = 1
Called with x = 101 y = 3
Called with x = 104 y = 5
Called with x = 109 y = 7
Called with x = 116 y = 9
125
```