Functional Programming with Python

Going Multi-Paradigm

A Tutorial at EuroPython 2019

8th July 2019

Basel, Switzerland

author: Dr.-Ing. Mike Müller
email: mmueller@python-academy.de
twitter: @pyacademy
version: 1.0

Python Academy - The Python Training Specialist

- Dedicated to Python training since 2006
- Wide variety of Python topics
- Experienced Python trainers
- Trainers are specialists and often core developers of taught software
- All levels of participants from novice to experienced software developers
- Courses for system administrators
- Courses for scientists and engineers
- Python for data analysis
- Open courses Europe-wide
- Customized in-house courses
- Python programming
- Development of technical and scientific software
- Code reviews
- Coaching of individuals and teams migrating to Python
- Workshops on developed software with detailed explanations

More Information: www.python-academy.com

Current Training Modules - Python Academy

As of 2019

Module Topic	Length (days)	In-house	Open
Python for Programmers	3	yes	yes
Python for Non-Programmers	4	yes	yes
Advanced Python	3	yes	yes
Python for Scientists and Engineers	3	yes	yes
Python for Data Analysis	3	yes	yes
Machine Learning with Python	3	yes	no
Professional Testing with Python	3	yes	yes
High Performance Computing with Python	3	yes	yes
Cython in Depth	2	yes	yes
Introduction to Django	4	yes	yes
Advanced Django	3	yes	yes
SQLAlchemy	1	yes	yes
High Performance XML with Python	1	yes	yes
Optimizing Python Programs	1	yes	yes
Python Extensions with Other Languages	1	yes	no
Data Storage with Python	1	yes	no
Introduction to Software Engineering with Python	1	yes	no
Introduction to wxPython	1	yes	no
Introduction to PySide/PyQt	1	yes	no
Overview of the Python Standard Library	1	yes	no
Threads and Processes in Python	1	yes	no
Windows Programming with Python	1	yes	no
Network Programming with Python	1	yes	no
Introduction to IronPython	1	yes	no

We offer on-site and open courses world-wide. We customize and extend training modules for your needs. We also provide consulting services such as code review, customized programming, and workshops.

More information: www.python-academy.com

Contents

1 Software **7**

1.1 Python 7

1.2 Libraries 7

1.3 IDE/Editor 7

2 Functional Programming Basics **8**

2.1 Programming Paradigms 8

2.2 Who uses it? 8

2.3 Features of Functional Programming 9

2.4 Advantages of Functional Programming 9

2.5 Disadvantages of Functional Programming 9

2.6 Python's Functional Features - Overview 9

3 Pure Functions **11**

3.1 No Side Effects 11

3.2 Side Effects 12

3.3 Functions are Objects 12

4 Callables and Functions in Python **13**

4.1 Callables 13

4.2 Closures 13

4.3 "Currying" 13

4.4 Partial Functions 14

4.5 Recursion 16

4.6 Lambda 18

4.7 Single Dispatch 18

4.8 Exercises 21

5 No Loops - `map`, `filter`, and `reduce` **23**

5.1 Processing iterables with `map` 23

5.2 Select from iterables with `filter` 24

5.3 Reductions of iterables with `reduce` 24

6 Operators as Functions **27**

6.1 Arithmetic Operators 27

6.2 Logical Operators 28

6.3 Attribute Access 29

6.4 Lookup 30

6.5 Exercises 31

7 Comprehensions **32**

7.1 Simple 32
7.2 Nested 32
7.3 Dictionary comprehensions 33
7.4 Set comprehensions 34
7.5 Exercises 34
8 Immutable Data Types 35
8.1 Tuples Instead of Lists 35
8.2 Frozen Sets 35
8.3 Not Only Functional 35
8.4 Reduce Side Effects by Moving Initializing into `__init__()` 36
8.5 Make Data Immutable 36
8.6 Stepwise Freezing and Thawing 37
8.7 Immutable Data Structures - Counter Arguments 38
9 Iterators 39
9.1 Itertools 39
9.1.1 Infinite iterators 41
9.1.2 Iterators terminating on the shortest input sequence 41
9.1.3 Combinatoric iterators 42
9.2 Exercises 42
10 More Itertools 44
10.1 Implementation Recipes 44
10.2 New Recipes 44
10.3 Exercises 47
11 Toolz 49
11.1 A functional library 49
11.2 Currying 49
11.3 Function Composition 51
11.4 Grouping 52
11.5 Exercises 54

1 Software

1.1 Python

Please install Python 3.7

Older Python version should, but may need some adjustments, as this tutorial uses f-strings.

1.2 Libraries

Please install `more-itertools`:

```
pip install more_itertools
```

or:

```
conda install -c conda-forge more_itertools
```

Please install `toolz`:

```
pip install toolz
```

or:

```
conda install -c conda-forge toolz
```

1.3 IDE/Editor

You are free to use any IDE or editor you are comfortable with. I will use JupyterLab because it makes a great teaching tool.

You can install it with `pip`:

```
pip install jupyterlab
```

or `conda`:

```
conda install -c conda-forge jupyterlab
```

2 Functional Programming Basics

2.1 Programming Paradigms

WikiPedia defines programming paradigms as follows:

> ***Programming Paradigms***
>
> Common programming paradigms include:
>
> **imperative** in which the programmer instructs the machine how to change its state,
>
> - **procedural** which groups instructions into procedures,
> - **object-oriented** which groups instructions together with the part of the state they operate on,
>
> **declarative** in which the programmer merely declares properties of the desired result, but not how to compute it
>
> - **functional** in which the desired result is declared as the value of a series of function applications,
> - **logic** in which the desired result is declared as the answer to a question about a system of facts and rules,
> - **mathematical** in which the desired result is declared as the solution of an optimization problem

Source: https://en.wikipedia.org/wiki/Programming_paradigm

While there are more paradigms (read the WikiPdi page cited above for many more paradigms), the ones above and the used systematic can be considered the most important. The imperative paradigm is by far the most used. Most of the commonly used programming languages such as C, C++, Java, Python, Javascript as well as newer, not (yet) that much used ones such as Swift, Go, Rust support this paradigm as the main or even only approach. Python fully supports the imperative paradigm by providing all feature for both procedural as well as object-oriented programming. Based on how typical open-source Python programs look like, most Python users predominately use this paradigm. This is totally fine, because all programming tasks can be solved with it. This is often referred as being Turing complete or computationally universal.

So why use a different paradigm? Turing completeness does not say anything how convenient, efficient, or error-resistant programming in a certain way will be. Declarative programming, if used for the right problems and the right way, can help get better programs in several aspects. Functional programming is likely the most use paradigm among the declarative one.

In addition to imperative programming support, Python has features that can be, at least partially, considered functional. Unlike languages such as Haskell, it is by no means purely functional. Nevertheless, these functional features can be used to produce better software. Python's strength is its multi-paradigm approach that allows to apply the most useful paradigm to a certain problem. Therefore, Python programs will be rarely written exclusively in a functional style.

Functional programming has a long history. The second oldest high-level programming language (FORTRAN was the first) still in common use is Lisp. It was first resealed in 1958.

In more recent times, languages such as Haskell, F#, Scala, and Erlang offer good support for functional programming, favor it as the preferred paradigm or are even purely functional.

2.2 Who uses it?

Functional programming is an established technique in certain industries such as:

- Trading
- Algorithmic development
- Telecommunication (for concurrency)

2.3 Features of Functional Programming

The declarative approach is realized by several features such as:

- Everything is a function
- *Pure* functions without side effects
- Immutable data structures
- Preserve state in functions (closure)
- Recursion instead of loops / iteration
- Lazy evaluation
- No global state

Not all listed features are restricted to functional programming and some may be used for programming with other paradigms. Depending on the actual language, there are more features that facilitate the functional approach.

2.4 Advantages of Functional Programming

These are some advantages of functional programming:

- Absence of side effects **can** make your programs more robust
- Programs tend to be more modular and come typically in smaller building blocks
- Better testable - a call with same parameters always returns same result due to no global state (except when intended such as random number generation or IO)
- Focus on algorithms
- Conceptional fit with parallel / concurrent programming
- Live updates - Install new release while running (Erlang)

2.5 Disadvantages of Functional Programming

If there would be only advantages, this paradigm should be the predominant one. But there also some disadvantages:

- Solutions to the same problem can look very different from procedural / object-oriented ones
- Finding good developers can be hard
- Not equally useful for all types of problems
- Input/output are side effects and need special treatment
- Recursion is "an order of magnitude more complex" than loops / iteration
- Immutable data structures may increase run times (if the implementation does not helps here)

2.6 Python's Functional Features - Overview

Python comes with some features that can help to write programs in a functional style:

- Pure functions (sort of, can always be broken)
- Closures - hold state in functions
- Functions are objects
- Decorators
- Immutable data types
- Lazy evaluation - generators
- List (dictionary, set) comprehensions
- Modules: `functools`, `itertools`, `operator`
- `lambda`, `map`, `filter`
- Recursion - try to avoid, recursion limit has a reason

In addition, there are several external libraries:

- more-itertools (about 88 new functions)
- Toolz with `itertoolz`, `functoolz`, and `itertoolz`
- CyToolz - Toolz implemented with Cython
- fn.py
- funcy
- Pyrsistent
- effect

There is also the new programming language Coconut that compiles to Python bytecode. It has many Haskell-like features and can use all existing Python libraries.

Also some popular libraries provide functional features. NumPy has universal functions. These functions can be, to some degree, considered a mapping for multiple dimensions. Furthermore, universal functions have the functions such as `accumulate()` and `reduce()` that work in a functional style on a whole NumPy array. The very popular library pandas uses NumPy internally. It also uses some functional features. For example, per default, many functions return new DataFrames instead of modifying the original DataFrame. As with NumPy, pandas program try to avoid loops in Python. This is (somewhat) similar to the no-loops approach of some pure functional languages.

3 Pure Functions

3.1 No Side Effects

Pure functions don't have sides effect. They only produce a return value. This function looks pure at first glance:

```
def do_pure(data):
    """Return copy times two.
    """
    return data * 2
```

This works for a number:

```
>>> do_pure(10)
20
```

as well as for a list:

```
>>> L = [1, 2, 3]
>>> res = do_pure(L)
>>> res
[1, 2, 3, 1, 2, 3]
>>> L
[1, 2, 3]
```

But there is no guarantee that there is no side effect. One way to break the code is this bad implementation of a list that counts how many times the special method `__mul__()` was called. Instead of using an instance variable, it modifies a global variable. This is typically a bad idea anyway. The upper case name `GLOBAL_LIST` indicates by convention that this should be a constant. This implementation:

```
"""Bad list that modifies global state when not expected.
"""


GLOBAL_LIST = []

class BadList(list):
    """Bad list with "non-pure" behavior for `*`.
    """

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.counter = 0

    def __mul__(self, factor):
        """Multiplication with integer and with **SIDE EFFECT**.
        """
        GLOBAL_LIST.append(self.counter)
        self.counter += 1
        return super().__mul__(factor)
```

leads to undesired side effects:

```
>>> GLOBAL_LIST
[]
>>> do_pure(bad_list)
[1, 2, 3, 1, 2, 3]
>>> GLOBAL_LIST
[0]
```

There is **no guarantee of pureness** in Python. As many other good-style recommendation pure functions can only be "guaranteed" by convention.

3.2 Side Effects

Side effects are common in functions. A function like this:

```
def do_side_effect(my_list):
    """Modify list appending 100.
    """
    my_list.append(100)
```

can make sense, but does not play well with the functional programming approach.

3.3 Functions are Objects

In Python pretty much everything is an object. Functions are no exception to this rule. These two functions:

```
def func1():
    return 1

def func2():
    return 2
```

can be stored in a dictionary:

```
>>> my_funcs = {'a': func1, 'b': func2}
>>> my_funcs['a']()
1
>>> my_funcs['b']()
2
```

just like any other object. This enables higher-order functions, i.e functions that can use functions as arguments or return functions as result.

4 Callables and Functions in Python

4.1 Callables

There are several "function-like" objects in Python, the "callables". These are callables:

- Objects with `__call__()` special method
- Built-in Functions
- Self-defined functions

As seen before, instances with special method can have state and therefore can easily break the assumption of a pure function.

4.2 Closures

The functions-are-objects features allows to define functions insides other functions and return them:

```
def outer(outer_arg):
    """
    A "function factory"
    """
    def inner(inner_arg):
        """Produced function
        """
        return inner_arg + outer_arg
    return inner
```

Now, `outer()` creates a new function:

```
>>> func = outer(12)
>>> func(5)
17
```

The value 12 is stored in a so-called closure:

```
>>> func.__closure__
(<cell at 0x102834b28: int object at 0x10040f640>,)
>>> func.__closure__[0]
<cell at 0x102834b28: int object at 0x10040f640>
>>> func.__closure__[0].cell_contents
12
```

This feature is useful for creating decorates.

4.3 "Currying"

Based on the mathematical principle of lambda calculus, the technique of currying is important for (pure) functional programming. For example, the purely functional language Haskell (named after Curry Haskell) makes ample use of this principle.

Essentially, currying is converting a single function of *n* arguments into *n* functions with a single argument each. Haskell is using it to turn all multi-argument functions in a series of single-argument functions.

In Python the closure works well for currying. This function with two arguments:

```
def func(x, y):
    """Simple function, called with two arguments
    """
    return x + y
```

can be replaced with a function that takes only one argument:

```
def func_curry(x):
    """Function, called with first argument
    """
    def inner(y):
        """Function, called with second argument
        """
        return x + y
    return inner
```

Now, this call:

```
>>> func(2, 3)
5
```

is equivalent to using our curried version:

```
>>> func_curry(2)(3)
5
```

Of course this is not restricted to two arguments. Adding another function definition with one more indention level can add another argument.

4.4 Partial Functions

What can you do with currying? One possibility is partial application. Module `functools` offers the function `partial` that makes this very easy. Using our function `func` from above, we can create a new function that has a "frozen" value for `y`:

```
>>> from functools import partial
>>> add_two = partial(func, y=2)
```

the resulting function takes only one argument:

```
>>> add_two(3)
5
```

Of course, you can do this in one line to come closer to our currying example from above:

```
>>> partial(func, y=2)(3)
5
```

You can use partial application for functions that take more than two arguments:

```
>>> import functools
>>> def func(a, b, c):
...     return a, b, c
...
```

"freezing" the first argument via a positional argument:

```
>>> p_func = functools.partial(func, 10)
>>> p_func(3, 4)
10 3 4
```

better use keyword arguments when freezing two arguments:

```
>>> p_func = functools.partial(func, a=10, b=12)
>>> p_func(3)
10 12 3
```

A good example for partial application is defining your own `print` function.

Starting wit a list:

```
>>> L = list(range(2, 11))
>>> print(L)
[2, 3, 4, 5, 6, 7, 8, 9, 10]
```

we can also print the single numbers, using the * syntax:

```
>>> print(*L)
2 3 4 5 6 7 8 9 10
```

and change the separator:

```
>>> print(*L, sep=', ')
2, 3, 4, 5, 6, 7, 8, 9, 10
```

With `partial` we can freeze the separator:

```
>>> print_comma_sep = partial(print, sep=', ')
>>> print_comma_sep(*L)
2, 3, 4, 5, 6, 7, 8, 9, 10
```

Be aware of Python's mutable default parameters

Be careful with mutable arguments. From this simple function:

```
>>> L = list(range(2, 11))
...
... def show(x):
...     print(x)
...
```

We can make a new function that hard-wires the list:

```
>>> show_list = partial(show, x=L)
>>> show_list()
[2, 3, 4, 5, 6, 7, 8, 9, 10]
```

But when you change the list:

```
>>> L.append(11)
```

This have an effect on the new function:

```
>>> show_list()
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

One way to avoid this behavior for this particular case is to supply the list elements as single arguments:

```
>>> L = list(range(2, 11))
>>> print_list = partial(print, *L, sep=', ')
>>> print_list()
2, 3, 4, 5, 6, 7, 8, 9, 10
```

Now, appending to this list:

```
>>> L.append(11)
L.append(11)
```

does change the printed list:

```
>>> print_list()
2, 3, 4, 5, 6, 7, 8, 9, 10
```

4.5 Recursion

Recursion is a very common functional feature. In fact, Haskell does not support typical loops, but requires the programmer to use recursion where Python programmers would use loops.

While Python supports recursion, it is often not a good way to solve a problem.

This a simple loop:

```
def loop(n):
    """Loop approach
    """
    res = 0
    for _ in range(n):
        res += 1
    return res
```

and this is the equivalent with recursion:

```
def recurse(n, res=0):
    """Recursion approach
    """
    if n > 0:
        res = recurse(n - 1, res + 1)
    return res
```

There are considerable differences in run times:

```
>>> n = 10
>>> t_loop = %timeit -o loop(n)
681 ns ± 9.03 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
>>> t_rec = %timeit -o recurse(n)
1.67 µs ± 43.6 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
>>> t_rec.average / t_loop.average
2.4559734806170805
```

The recursive version is but 2.5 x slower for 10 loops. Increasing the loops will increase the difference:

```
>>> n = 1_000
>>> t_loop = %timeit -o loop(n)
54.5 µs ± 621 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
>>> t_rec = %timeit -o recurse(n)
228 µs ± 18.9 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
>>> t_rec.average / t_loop.average
4.179608707159303
```

Python has a default recursion limit of 3000. We can increase it:

```
>>> import sys
>>> sys.setrecursionlimit(110_000)
>>> t_rec = %timeit -o recurse(n)
```

and now can continue increasing the loops:

```
>>> n = 10_000
>>> t_loop = %timeit -o loop(n)
588 µs ± 14.7 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
3.18 ms ± 41.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
>>> t_rec.average / t_loop.average
5.4059613373849
```

```
>>> n = 20_000
>>> t_loop = %timeit -o loop(n)
1.19 ms ± 8.87 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
>>> t_rec = %timeit -o recurse(n)
9.49 ms ± 160 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
>>> t_rec.average / t_loop.average
8.001911632781033
```

The timing did not finish for 30,000 recursions on a decent laptop.

4.6 Lambda

Python supports anonymous functions. They are rather limited and there was a serious discussion among Python core developers to remove `lambda` from Python 3 altogether. But it is still here.

Lambdas are restricted to expressions. No statements are allowed in the functions body. This means no direct exception handling, no assignment with =, and no temporary variables.

Lambdas are especially useful for short and simple functions that are for callbacks. For example, this function:

```
def use_callback(callback, arg):
     return callback(arg)
```

expects a function as its first argument. This makes a good candidate for a lambda:

```
>>> use_callback(lambda arg: arg * 2, 10)
20
```

The lambda construct is not essential. It may be preferable to add a few extra lines and write a function with a name and a docstring:

```
def double(arg):
   """Double the argument.
   """
   return arg *2
```

This makes the use as callback arguably more readable:

```
>>> use_callback(double, 10)
20
```

Using a good and telling name is important here.

4.7 Single Dispatch

Since version 3.4 Python supports single dispatch. Statically typed languages can support multiple dispatch at the compiler level. Multiple functions with the same name but differently typed parameters can do (slightly) different things.

Because Python is dynamically typed and all parameters of functions at definition time are just objects, a multiple dispatch is not possible without extra work. The Python developers decided to only support single

dispatch to keep the usage simple. This means that the type of the first argument decides what the function does in particular.

Let's look at an example. We have a function that shows some information about an object:

```
def show_object_undispatch(obj):
    """Show information about an object
    """
    print(f'Type: {type(obj).__name__}')
    print(f'Value: {obj}')
```

using this function, provides a nicely formated output:

```
>>> show_object_undispatch(2)
Type: int
Value: 2
```

This works also for lists:

```
>>> show_object_undispatch(list(range(2, 11)))
Type: list
Value: [2, 3, 4, 5, 6, 7, 8, 9, 10]
```

But it generates a lot of output for large lists:

```
>> show_object_undispatch(list(range(2, 100)))
Type: list
Value: [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, # output cut off
```

(The output was cut off for a better display. It goes on to 99.) So it would make sense to implement a different function for lists. It would be great if the user could use the same function for any object and the function would decide how to display the object based on its type. Of course it is possible to use `instance` checks inside the function, but the single dispatch feature makes this much easier.

We decorate our function with `@singledispatch`:

```
@singledispatch
def show_object(obj):
    """Show information about an object
    """
    print(f'Type: {type(obj).__name__}')
    print(f'Value: {obj}')
```

and use this decorator to register a new function `_show_object_list()` (The name does not matter and often _ is used for such a decorated function.):

```
@show_object.register(list)
def _show_object_list(obj):
    """Show information about a list
    """
    print(f'Type: {type(obj).__name__}')
    if len(obj) > 10:
```

```
        print(f'Value: {repr(obj[:3])[:-1]} ... {repr(obj[-3:])[1:]}')
    else:
        print(f'Value: {obj}')
    print(f'Length: {len(obj)}')
```

Our display for an integer looks the same as before:

```
>>> show_object(2)
Type: int
Value: 2
```

Calling it for our short list displays the list length in addition to the other information:

```
>>> show_object(list(range(2, 11)))
Type: list
Value: [2, 3, 4, 5, 6, 7, 8, 9, 10]
Length: 9
```

Finally, the output for our large list looks much nicer:

```
>>> show_object(list(range(2, 100)))
Type: list
Value: [2, 3, 4 ... 97, 98, 99]
Length: 98
```

Of course, this is not restricted to one type. We can also apply the decorator multiple times:

```
@show_object.register(tuple)
@show_object.register(list)
def _show_object_sequence(obj):
    """Show information about a list, tuple, and set
    """
    print(f'Type: {type(obj).__name__}')
    if len(obj) > 10:
        print(f'Value: {repr(obj[:3])[:-1]} ... {repr(obj[-3:])[1:]}')
    else:
        print(f'Value: {obj}')
    print(f'Length: {len(obj)}')
```

The result for a list is the same:

```
>>> show_object(list(range(2, 100)))
Type: list
Value: [2, 3, 4 ... 97, 98, 99]
Length: 98
```

Now, it also works for tuples:

```
>>> show_object(tuple(range(2, 100)))
Type: tuple
```

```
Value: (2, 3, 4 ... 97, 98, 99)
Length: 98
```

As with any decorator, we are not restricted to the application with @, but can register our function later:

```
def show_string(obj):
    """Show information about a string
    """
    print('String')
    print('======')
    print(f'Type: {type(obj).__name__}')
    if len(obj) > 10:
        print(f'Value: {repr(obj[:3])[:-1]} ... {repr(obj[-3:])[1:]}')
    else:
        print(f'Value: {obj}')
    print(f'Length: {len(obj)}')


show_object.register(str)(show_string)
```

This allows to import a function from another module and applying the decorator without touching the source code of the imported module. Now, our display works for strings too:

```
>>> show_object('a' * 100)
String
======
Type: str
Value: 'aaa ... aaa'
Length: 100
```

The original types still work as before:

```
>>> show_object(tuple(range(2, 100)))
Type: tuple
Value: (2, 3, 4 ... 97, 98, 99)
Length: 98
```

4.8 Exercises

1. Rewrite the function:

   ```
   def func(a, b, c):
       return a * b * c
   ```

   that is called with three arguments:

   ```
   >>> func(3, 2, 4)
   24
   ```

   The result should be called with a single argument, return a new function that in turn can be called with one argument and returns a final function that also takes one argument, i.e.

```
>>> func(3)(2)(4)
24
```

2. Using `functools.partial`, create a new function `openw` from the built-in `open` that opens a file in write mode, i.e. this code should work:

```
with openw('out.txt') as fobj:
   fobj.write('text\n')
```

5 No Loops - `map`, `filter`, and `reduce`

Functional programming provides several basic patterns that help to solve typical imperative loop problems. The most common ones are probably `map`, `filter`, and `reduce`.

5.1 Processing iterables with `map`

`map` can apply, i.e, map, a function to each element of an iterable.

This list of integers:

```
>>> L = list(range(2, 11))
>>> L
[2, 3, 4, 5, 6, 7, 8, 9, 10]
```

can be converted into a list of floats:

```
>>> m = map(float, L)
>>> m
<map at 0x103109908>
```

The result is a map object, which is an iterator:

```
>>> type(m) is type(iter(m))
True
```

Therefore, it supports `next(map_object)`:

```
>>> next(m)
2.0
```

as well as conversion (materialization) into an object that holds all elements:

```
>>> list(m)
[3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]
```

Note the missing `2.0`, which was already consumed by `next()`. The next attempt to convert the map object returns an empty list:

```
>>> list(m)
[]
```

The object has been exhausted.

Pythonic programming favors list comprehensions over the to use `map()`:

```
>>> [float(x) for x in L]
[2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]
```

or, to stay lazy, use a generator expression:

```
>>> (float(x) for x in L)
<generator object <genexpr> at 0x10306aed0>
```

5.2 Select from iterables with `filter`

The built-in `filter` works in a similar way as map. But instead of converting all elements with a function, it keeps only the elements for which the function returns `True`.

The result is also an iterator. In the following examples this iterator is converted into a list to visualize its content.

Starting with a list:

```
>>> L
[2, 3, 4, 5, 6, 7, 8, 9, 10]
```

and a simple function that returns `True` if its argument is greater than five:

```
>>> def greater5(x):
...     return x > 5
```

Calling `filter` with the function as first and the list as second argument, only elements that are greater than 5 will end up in the result:

```
>>> list(filter(greater5, L))
[6, 7, 8, 9, 10]
```

Since the function is very simple, it make a good candidate for `lambda`:

```
>>> list(filter(lambda x : x > 5, L))
[6, 7, 8, 9, 10]
```

Again, it is considered more pythonic to use a list comprehension with a condition:

```
>>> [x for x in L if x > 5]
[6, 7, 8, 9, 10]
```

or, to stay lazy, use the corresponding generator expression:

```
>>> (x for x in L if x > 5)
<generator object <genexpr> at 0x10306ae58>
```

5.3 Reductions of iterables with `reduce`

This is what the docstrings says about `reduce`

> ***Definition of*** `reduce`
>
> Apply a function of two arguments cumulatively to the items of a sequence, from left to right, so as to reduce the sequence to a single value.

After importing `reduce`:

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

and defining a function that takes two arguments:

```
>>> def add(x, y):
...     return x + y
```

our list is reduced into one number, which is the sum of all list elements:

```
>>> reduce(add, L)
54
```

Again, this simple functions suggests the use of `lambda`:

```
>>> reduce(lambda x, y: x + y, L)
54
```

Of course, the built-in `sum` already exits:

```
>>> sum(L)
54
```

Multiplication of all numbers seems a bit more useful:

```
>>> reduce(lambda x, y: x * y, L)
3628800
```

One way to understand `reduce` is to think of it as reduction of dimensionality. The examples above reduce a one-dimensional list into scalar. It is also possible to reduce a two-dimensional list (Actually, it is a nested list of lists, but can be conceptually a a two-dimensional list.):

```
>>> LL = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

into a one-dimensional list:

```
>>> reduce(add, LL)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The result does not have to be a list:

```
>>> reduce(add, ((1, 2, 3),))
(1, 2, 3)
```

Of course this strange function:

```
>>> def wrong(x, y):
...     return [[[x, y]]]
```

also works with `reduce`:

```
>>> reduce(wrong, L)
[[[[[[[[[[[[[[[[[[[[[[[2, 3]]], 4]]], 5]]], 6]]], 7]]], 8]]], 9]]], 10]]]
```

This is more an expansion than reduction. Probably not what the user expects. Better apply `reduce` in the intended way.

6 Operators as Functions

The operator module of the standard library provides functions that represent Python's operators. These operators maybe grouped into:

- Arithmetic
- Logical
- Attribute access and
- Lookup

operators.

Many operator functions, such as `add`, have a second version with a name that looks like a special method name, i.e `__add__`. The underscore versions exists for backward compatibility only and should not be used.

6.1 Arithmetic Operators

For arithmetic operators, such as `+`, `-`, `*`, and `/` it can be useful to store the corresponding functions in a data structure such as a dictionary. This is a dictionary, that maps some arithmetic operators to functions:

```
MATH_OPS = {
    '+': operator.add,
    '-': operator.sub,
    '*': operator.mul,
    '/': operator.truediv,
    }
```

This mapping can be used to evaluate an equation:

```
def eval_equation(equation):
    """
    Evaluate the equation
    """
    number1, opr, number2 = equation.split()
    return MATH_OPS[opr](float(number1), float(number2))
```

The main program adds a simple user interface:

```
def main():
    """Calculator for simple arithmetic operations
    """
    msg = """
        Please enter an equation in the form: <num> <op> <num>
        where:
            <num> is a number such 10 or 4.5
            <op> an operator, one of +, - *, or /
        Examples:
        1 + 1
        10.5 * 3
        12.5 / 2
        Note: Spaces around the operator are needed.
```

```
        Your equation: """
    equation = input(dedent(msg))
    print(equation, '=', eval_equation(equation))


if __name__ == '__main__':
    main()
```

running this program:

```
python calculator.py
```

generates this output:

```
Please enter an equation in the form: <num> <op> <num>
where:
    <num> is a number such 10 or 4.5
    <op> an operator, one of +, - *, or /
Examples:
1 + 1
10.5 * 3
12.5 / 2

Your equation: 3 + 5
3 + 5 = 8.0
```

6.2 Logical Operators

These are some examples for logical operators.

Greater than:

```
>>> 40 > 30
True
>>> operator.gt(40, 30)
True
```

`is`:

```
>>> 1 is 1
True
>>> operator.is_(1, 1)
True
```

Bool:

```
>>> bool(0)
False
>>> operator.truth(0)
False
```

```
>>> bool(10)
>>> operator.truth(10)
True
```

`in`:

```
>>> operator.contains([1, 2, 3], 2)
True
```

6.3 Attribute Access

An interesting function is `operator.attrgetter()` that represents the `inst.attr` access with function.

This simple class:

```
class A:

    def __init__(self, value1, value2):
        self.value1 = value1
        self.value2 = value2

    def __repr__(self):
        return f'A({self.value1}, {self.value2})'
```

shows its two attributes in its `repr`:

```
>>> a = A(10, 20)
>>> a
A(10, 20)
```

Now, "freeze" `a.value1` in a function `g1()`:

```
>>> g1 = operator.attrgetter('value1')
>>> g1(a)
10
```

This works also for multiple attributes:

```
>>> g21 = operator.attrgetter('value2', 'value1')
>>> g21(a)
(20, 10)
```

This can be useful for callbacks. The built-in `sorted` takes a callback function with `key=`. With this list:

```
>>> data = [A(1, 2), A(10, 0), A(2, 3)]
```

and the right function:

```
>>> g2 = operator.attrgetter('value2')
```

sorting by `value2` is simple:

```
>>> sorted(data, key=g2)
[A(10, 0), A(1, 2), A(2, 3)]
```

This would be the `lambda` equivalent:

```
>>> sorted(data, key=lambda x: x.value2)
[A(10, 0), A(1, 2), A(2, 3)]
```

6.4 Lookup

In a similar way, `operator.itemgetter()` is the function for `obj[key]` or `obj[index]`. This dictionary:

```
>>> d = {'a': 100, 'b': 200, 'c': 300}
```

allows to access its values by keys:

```
>>> d['a']
100
```

The same can be achieved with `operator.itemgetter()`:

```
>>> ga = operator.itemgetter('a')
>>> ga(d)
100
```

Likewise, this works for sequences. Access with `[]`:

```
>>> L = list(range(2, 11))
>>> L[3]
5
```

or with a function:

```
>>> g0 = operator.itemgetter(3)
>>> g0(L)
5
```

Again this can be useful for sorting. This dictionary:

```
>>> d = {'b': 300, 'a': 200, 'c': 100}
```

allows sorting of its items by key:

```
>>> sorted(d.items())
[('a', 200), ('b', 300), ('c', 100)]
```

This is equivalent to:

```
>>> sorted(d.items(), key=operator.itemgetter(0))
[('a', 200), ('b', 300), ('c', 100)]
```

This sorts by the dictionary value:

```
>>> sorted(d.items(), key=operator.itemgetter(1))
[('c', 100), ('a', 200), ('b', 300)]
```

6.5 Exercises

1. Write new function `my_sum` that adds the numbers in a list in one line as a `lambda``function`. Use ``reduce` as well as the corresponding function from the module `operator`.

7 Comprehensions

7.1 Simple

The principle comes from the functional language Haskell but integrates very well into Python. List comprehensions can make our lives easier:

```
>>> [x for x in range(20) if 3 < x  < 15]
[4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

This replaces the old `filter` function:

```
>>> list(filter(lambda x: 3 < x  < 15, range(20)))
[4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

We can also use methods on objects:

```
>>> [x.upper() for x in 'abcdef']
['A', 'B', 'C', 'D', 'E', 'F']
```

or functions:

```
>>> [abs(x) for x in (1, 2, -4, 5, -8)]
[1, 2, 4, 5, 8]
```

This replaces the old `map`:

```
>>> list(map(abs, (1, 2, -4, 5, -8)))
[1, 2, 4, 5, 8]
```

Ranges in Python 2

In Python 3 `range()` creates a new range object. But in Python 2 `range()` creates a new list. In Python 2, you can use `xrange()` instead of `range()` for large ranges to save memory. xrange objects are (for most purposes) equivalent to the Python 3 range objects. A simple `range = xrange` at the beginning of a script might be good enough. For larger projects, it is recommended to use helper libraries such as "python-future.org" to get the Python 3 `range` in Python 2.

7.2 Nested

There is no limit to nesting list comprehensions:

```
>>> nested = [(x, y) for x in range(5) for y in range(10,15)]
```

This is equivalent to:

```
>>> res = []
>>> for x in range(5):
...     for y in range(10, 15):
...         res.append((x, y))
...
>>> res == nested
True
```

This goes three levels deep:

```
>>> deeply_nested = [x * y * z for x in range(10) for y in range(10)
...                  for z in range(10)]
```

This is equivalent to:

```
>>> res = []
>>> for x in range(10):
...     for y in range(10):
...         for z in range(10):
...             res.append(x * y * z)
...
>>> deeply_nested == res
True
```

7.3 Dictionary comprehensions

Python offers dictionary comprehensions:

```
>>> {x: x.upper() for x in 'abcdefg'}
{'a': 'A', 'c': 'C', 'b': 'B', 'e': 'E', 'd': 'D', 'g': 'G', 'f': 'F'}
```

Alternatively, it is possible to create tuples and use the built-in function `dict`:

```
>>> dict((x, x.upper()) for x in 'abcdefg')
{'a': 'A', 'c': 'C', 'b': 'B', 'e': 'E', 'd': 'D', 'g': 'G', 'f': 'F'}
```

Side note: In Python versions older than 2.7, the latter is the only option, because dictionary comprehensions were introduced in Python 2.7.

Note that we can use a generator expression and can drop the additional parenthesis.

We can swap keys and values:

```
>>> {v:k for k, v in dic.items()}
{'A': 'a', 'C': 'c', 'B': 'b', 'E': 'e', 'D': 'd', 'G': 'g', 'F': 'f'}
```

The second approach with `dict` is just a little longer:

```
>>> dict((v, k) for k, v in dic.items())
{'A': 'a', 'C': 'c', 'B': 'b', 'E': 'e', 'D': 'd', 'G': 'g', 'F': 'f'}
```

7.4 Set comprehensions

Set comprehensions are very similar to list comprehensions, just replace the `[]` with `{}`:

```
>>> myset = {1, 2, 3, 4, 5, 6, 7, 8}
>>> myset
{1, 2, 3, 4, 5, 6, 7, 8}
>>> {x * 2 for x in myset if 2 < x < 7}
{8, 10, 12, 6}
```

There is also a second way, using the built-in function `set` and a a generator expression:

```
>>> myset = set([1, 2, 3, 4, 5, 6, 7, 8])
>>> set(x * 2 for x in myset if 2 < x < 7)
set([8, 10, 12, 6])
```

7.5 Exercises

1. Use a list comprehension to convert a list of numbers from 10 to 50 into their squares if the numbers are even.
2. Write a nested list comprehension that iterates through the lists x, y and z each holding floats that represent coordinates. The comprehension should return a list of tuples. Each such tuple should contain three values, one from x, one from y, and one form z for all values of x that are within given limits you provide.
3. (a) Write a dictionary comprehension that goes through a list of numbers and uses the square of the number as key and the number itself as value. (b) Solve the same task with a generator expression that is converted into a dictionary.
4. (a) Write a set comprehension that creates a set from a list of numbers turning negative numbers (that should be in the list) into positive numbers. (b) Solve the same task with a generator expression that is converted into a set.

8 Immutable Data Types

Pure functional programming languages often work only with immutable data types. Python gets lots of its power from mutable data types such as lists and dictionaries. But sometimes going with immutable data types can make programs better.

8.1 Tuples Instead of Lists

A mutable list can be easily converted into an immutable set:

```
>>> my_list = list(range(10))
>>> my_list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> my_tuple = tuple(my_list)
>>> my_tuple
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

But using tuples instead of lists can contradicts the usage recommendation for these data types:

- Lists == elements of the same kind
- Tuple == "named" elements

For example, a table with values of the same type in each column would be represented by:

- Columns with values of same type as lists
- Rows with mixed types as tuples

But: "Although practicality beats purity."

8.2 Frozen Sets

The equivalent "list and tuple" is "set and frozenset":

```
>>> my_set = set(range(5))
>>> my_set
set([0, 1, 2, 3, 4])
>>> my_frozenset = frozenset(my_set)
>>> my_frozenset
frozenset([0, 1, 2, 3, 4])
```

As a side effect, frozensets can be used as dictionary keys.

8.3 Not Only Functional

Pure functional programs are essentialy impossible to implement if libraries are used, that use "non-pure" features. Therefore, functional program parts can be combined with procedural and object-oriented program parts. It is important to choose the right tool for the task at hand. Over time the developer needs to get a feeling where a functional approach can be beneficial. For example, IO operations and use of large OOP GUI libraries are not suitable for the functional approach. On the other hand, algorithmic program parts that process data in distinct steps can be a good candidate for going functional.

8.4 Reduce Side Effects by Moving Initializing into `__init__()`

While side effects can be useful, they may be reduced or totally avoided in some places. This class initializes all its attributes in the `__init__()`, which can help to reduce side effects because there is no need to initialize anything after instantiation:

```
class InitOnly:
    """Example for init-only definitions.
    """
    def __init__(self, count):
        self.attr1 = self._make_attr(count)
        self.attr2 = self._make_attr(count + 10)

    @staticmethod
    def _make_attr(count):
        """Do many things to create an attribute.
        """
        attr = []
        for x in range(count):
            # do something actual useful here
            attr.append(x)
        return tuple(attr)
```

To keep the `__init__()` short, the actual functionality is moved into a static method. BTW, when you use PyLint, you will get a warning message if you initialize and attribute outside of the `__init__()`.

8.5 Make Data Immutable

Mutable data structures are very useful for reading data. Converting them into immutable version can reduce the danger of unwanted modifications later in the live of the instance.

This reader class does the reading only in the `__init__()`:

```
class Reader:
    """Read some data
    """
    def __init__(self, file_name):
        self.data = self._read(file_name)

    @staticmethod
    def _read(file_name):
        """Return tuple of tuple of read data.
        """
        data = []
        with open(file_name) as fobj:
            for line in fobj:
                data.append(tuple(line.split()))
        return tuple(data)
```

Here a version with a one-line reader, which still kind of readable:

```
class ReaderOneLine:
    """Read some data
```

```
    """
    def __init__(self, file_name):
        self.data = self._read(file_name)

    @staticmethod
    def _read(file_name):
        """Return tuple of tuple of read data.
        """
        return tuple(tuple(line.split()) for line in open(file_name))
```

While this makes `data` itself immutable, the attribute can still be replaced altogether. The next technique can help here.

8.6 Stepwise Freezing and Thawing

It can be useful to prevent overriding attributes. This class offers two methods `freeze()` and `unfreeze()` to explicitly turn on and off the setting of new attributes:

```
"""
Instance freezer
"""


class Freezer:
    """
    Simple helper class to prevent changing of attributes on instances
    """
    def __init__(self):
        self._frozen = False

    def __setattr__(self, name, value):
        if name != '_frozen' and self._frozen:
            raise AttributeError(f'Cannot set attribute {name}')
        super().__setattr__(name, value)

    def freeze(self):
        """Turn of freezing
        """
        self._frozen = True

    def unfreeze(self):
        """Turn off freezing
        """
        self._frozen = False
```

A use case for this type of freezing can be legacy code. It might be useful to find out if and where attributes of an instance are actually assigned. In complex systems it can be interesting to detect if there unwanted attribute assignments.

This existing class:

```
class Old:
    """Existing class to frozen
```

```
    """
    def __init__(self, value):
        self.value = value

    def meth(self, factor):
        """Do something
        """
        return self.value * factor
```

can be replaced by a frozen class:

```
class OldFrozen(Old, Freezer):
    """Old class but frozen
    """
    def __init__(self, *args, **kwargs):
        Freezer.__init__(self)
        super().__init__(*args, **kwargs)
        self._frozen = True
```

Now, the attempt to assign a new attribute fails:

```
def test():
    """Test the freezing
    """
    old_frozen = OldFrozen(10)
    print(old_frozen.meth(3))
    # should raise an exception
    old_frozen.value = 12
```

```
>>>test()
30
...
AttributeError: Cannot set attribute value
```

8.7 Immutable Data Structures - Counter Arguments

Mutable data structures are really useful. On the other hand, some algorithms maybe difficult to implement with immutable data structures. In addition, it can be really inefficient to use immutable data structures due to repeated re-allocation of memory when immutable structures with nearly the same data are created repeatedly.

One of this anti pattern is string concatenation:

```
>>>  s += 'text'
```

While this case is optimized in CPython, it can make the run times explode for other implementations such as Jython or PyPy.

9 Iterators

Iterators are objects that have the methods `next` and `__iter__`:

```
>>> class CountDown(object):
...     def __init__(self, start):
...         self.counter = start + 1
...     def __next__(self):  # Python 3
...         self.counter -= 1
...         if self.counter <= 0:
...             raise StopIteration
...         return self.counter
...     next =  __next__  # Python 2
...     def __iter__(self):
...         return self
```

`__iter__` has to return the iterator itself, and `__next__` should return the next element and raise `StopIteration` when finished. In Python 2, `__next__` has to be named `next`. Now we can use our iterator:

```
>>> cd = CountDown(5)
>>> for x in cd:
...     print(x)
...
5
4
3
2
1
```

A sequence can be turned into an iterator using the built-in function `iter`:

```
>>> i = iter(range(5, 0, -1))
>>> next(i)
5
>>> next(i)
4
>>> i.next() # old way in Python 2 only
3
>>> next(i)
2
>>> next(i)
1
>>> next(i)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
StopIteration
```

9.1 Itertools

The `itertools` module in the standard library offers powerful functions that work with and return iterators.

After importing the module:

```
>>> import itertools as it
```

we can have an infinity iterator that starts at the beginning after reaching its end with `cycle`:

```
>>> cycler = it.cycle([1,2,3])
>>> next(cycler)
1
>>> next(cycler)
2
>>> next(cycler)
3
>>> next(cycler)
1
>>> next(cycler)
2
>>> next(cycler)
3
>>> next(cycler)
1
```

The function `counter` provides an infinite counter with an optional start value (default is zero):

```
>>> counter = it.count(5)
>>> next(counter)
5
>>> next(counter)
6
```

With `repeat`, we can construct a new iterator that also can be infinite:

```
>>> list(it.repeat(4, 2))
[4, 4]
>>> list(it.repeat(2, 4))
[2, 2, 2, 2]
>>> endless  = it.repeat(3)
>>> next(endless)
3
>>> next(endless)
3
>>> next(endless)
3
```

The variation `zip_longest` fills missing values:

```
>>> list(it.zip_longest([1,2,3], [4,5,6,7,8]))
[(1, 4), (2, 5), (3, 6), (None, 7), (None, 8)]
>>> list(it.zip_longest([1,2,3], [4,5,6,7,8], fillvalue=99999))
[(1, 4), (2, 5), (3, 6), (99999, 7), (99999, 8)]
```

Two or more iterables can be combined in one with `chain`:

```
>>> list(it.chain([1,2,3], [4,5,6]))
[1, 2, 3, 4, 5, 6]
```

To get only part of an iterable, we can use `islice` that works very similar to the slicing of sequences:

```
>>> list(it.islice(range(10), 5))
[0, 1, 2, 3, 4]
>>> list(it.islice(range(10), 5, 8))
[5, 6, 7]
>>> list(it.islice(range(10), 5, None))
[5, 6, 7, 8, 9]
>>> list(it.islice(range(10), 5, None, 2))
[5, 7, 9]
>>>
```

9.1.1 Infinite iterators

Iterator	Arguments	Results	Example
`count()`	`start, [step]`	start, start+step, start+2*step, …	`count(10) --> 10 11 12 13 14 ...`
`cycle()`	`p`	p0, p1, … plast, p0, p1, …	`cycle('ABCD') --> A B C D A B C D ...`
`repeat()`	`elem [,n]`	elem, elem, elem, … endlessly or up to n times	`repeat(10, 3) --> 10 10 10`

9.1.2 Iterators terminating on the shortest input sequence

Iterator	Arguments	Results	Example
`accumulate()`	`p [,func]`	p0, p0+p1, p0+p1+p2, …	`accumulate([1,2,3,4,5]) --> 1 3 6 10 15`
`chain()`	`p, q, …`	p0, p1, … plast, q0, q1, …	`chain('ABC', 'DEF') --> A B C D E F`
`chain.from_iterable()`	iterable	p0, p1, … plast, q0, q1, …	`chain.from_iterable(['ABC', 'DEF']) --> A B C D E F`
`compress()`	data, selectors	(d[0] if s[0]), (d[1] if s[1]), …	`compress('ABCDEF' [1,0,1,0,1,1]) --> A C E F`
`dropwhile()`	pred, seq	seq[n], seq[n+1], starting when pred fails	`dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1`
`filterfalse()`	pred, seq	elements of seq where pred(elem) is false	`filterfalse( lambda x: x%2, range(10)) --> 0 2 4 6 8`
`groupby()`	iterable[, key]	sub-iterators grouped by value of key(v)	

`islice()`	seq, [start,] stop [, step]	elements from seq[start:stop:step]	`islice('ABCDEFG', 2, None) --> C D E F G`
`starmap()`	func, seq	func(*seq[0]), func(*seq[1]), …	`starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000`
`takewhile()`	pred, seq	seq[0], seq[1], until pred fails	`takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4`
`tee()`	it, n	it1, it2, … itn splits one iterator into n	
`zip_longest()`	p, q, …	(p[0], q[0]), (p[1], q[1]), …	`zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-`

9.1.3 Combinatoric iterators

Iterator	Arguments	Results
`product()`	p, q, … [repeat=1]	cartesian product, equivalent to a nested for-loop
`permutations()`	p[, r]	r-length tuples, all possible orderings, no repeated elements
`combinations()`	p, r	r-length tuples, in sorted order, no repeated elements
`combinations_with_replacement()`	p, r	r-length tuples, in sorted order, with repeated elements
`product('ABCD', repeat=2)`		AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD
`permutations('ABCD', 2)`		AB AC AD BA BC BD CA CB CD DA DB DC
`combinations('ABCD', 2)`		AB AC AD BC BD CD
`combinations_with_replacement('ABCD', 2)`		AA AB AC AD BB BC BD CC CD DD

9.2 Exercises

1. Using functions from `itertools`, write a generator that creates an endless stream of numbers starting from a value given as argument with a step size of 5.
2. Rewrite the following code snippets using `itertools`.

```
x = 0
while True:
    x += 1
    print(x)

[1, 2, 3, 4, 5][2:]
[1, 2, 3, 4, 5][:4]
```

```
[1, 2, 3] + [4, 5, 6]
```

10 More Itertools

The package `more-itertools` extends the `iterools` module of the standard library.

10.1 Implementation Recipes

The Python's standard library documentation shows several interesting recipes for writing iterators for different purposes (http://docs.python.org/library/itertools.html#recipes). `more-itertools` implements these 27 recipes with some backward-compatible usability improvements.

Let's have a look at some interesting recipes.

After importing:

```
>>> from more_itertools import recipes
```

we can use `consume()` that consumes an iterator:

```
>>> i = iter('abcdefg')
>>> recipes.consume(i)
>>> list(i)
[]
```

Most of the time, it is more useful to consume only some iterator elements:

```
>>> i = iter('abcdefg')
>>> recipes.consume(i, 3)
>>> list(i)
['d', 'e', 'f', 'g']
```

There is also a solution for flattening a nested list:

```
>>> list(recipes.flatten([[1, 2, 3], [4, 5], [6, 7, 8, 9]]))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The function `take` returns first n elements of a an iterator as a list:

```
>>> i = iter('abcdefg')
>>> recipes.take(2, i)
['a', 'b']
```

10.2 New Recipes

In addition to the recipes from the standard library `more-itertools` provides 61 new recipes. These can be grouped into these categories:

- Grouping
- Lookahead and lookback
- Windowing
- Augmenting
- Combining

- Summarizing
- Selecting
- Combinatorics
- Wrapping

Let's have a look at some interesting recipes.

After importing:

```
>>> import more_itertools as more
```

an creating an iterator:

```
>>> i = iter(range(10, 51, 10))
```

we can use `spy` to see what will be next in the iterator and get a new iterator that hast the same state as the iterator before our spying:

```
>>> first, i = more.spy(i, 1)
```

This is first element in a list:

```
>>> first
[10]
```

The returned iterator looks the one before our interrogation:

```
>>> list(i)
[10, 20, 30, 40, 50]
```

We can also look at more than one element:

```
>>> i = iter(range(10, 51, 10))
>>> first3, i = more.spy(i, 3)
```

and see the first three elements:

```
>>> first3
[10, 20, 30]
```

Again, the returned iterator brings us back to the pre-syp state:

```
>>> list(i)
[10, 20, 30, 40, 50]
```

The sequence unpacking works as excepted:

```
>>> i = iter(range(10, 51, 10))
>>> (first, second, third), i = more.spy(i, 3)
```

Now we have the first three elements with their own names:

```
>>> first
10
>>> second
20
>>> third
30
```

Same situation for the iterator as before:

```
>>> list(i)
[10, 20, 30, 40, 50]
```

Spaying for more elements than the iterator can provide:

```
>>> i = iter(range(10, 51, 10))
>>> head, i = more.spy(i, 8)
```

The head contains all elements the iterator can produce:

```
>>> head
[10, 20, 30, 40, 50]
```

Of course, the returned iterator brings us back to the state before:

```
>>> list(i)
[10, 20, 30, 40, 50]
```

An empty iterator works too:

```
>>> i = iter([])
>> head, i = more.spy(i, 8)
```

And empty head:

```
>>> head
[]
```

and an empty iterator:

```
>>> list(i)
[]
```

are the result.

Starting with this data:

```
>>> data1 = list(range(2, 11))
>>> data2 = list('ABCDE')
```

the standard `zip` creates the combination of both, starting from the beginning and ending at with sorter of the both:

```
>>> list(zip(data1, data2))
[(2, 'A'), (3, 'B'), (4, 'C'), (5, 'D'), (6, 'E')]
```

`zip_offset` from `more-iterools` can take off sets for zipped data structures:

```
>>> list(more.zip_offset(data1, data2, offsets=(3, 1)))
[(5, 'B'), (6, 'C'), (7, 'D'), (8, 'E')]
```

This list of numbers is offset by 3, i.e. starting from 5 instead from 2. The list of strings is offset by 1, i.e. starting at B instead of A.

`zip_offset` also provides `zip_longest` behavior:

```
>>> list(more.zip_offset(data1, data2, offsets=(3, 1),
...                      longest=True))
...
[(5, 'B'), (6, 'C'), (7, 'D'), (8, 'E'), (9, None), (10, None)]
```

including a different fill value:

```
>>> list(more.zip_offset(data1, data2, offsets=(3, 1),
...                      longest=True, fillvalue='XXX'))
...
[(5, 'B'), (6, 'C'), (7, 'D'), (8, 'E'), (9, 'XXX'), (10, 'XXX')]
```

`chunked` shops an iterator into equal-sized list:

```
>>> list(more.chunked(range(2, 11), 4))
[[2, 3, 4, 5], [6, 7, 8, 9], [10]]
```

Because the 9 elements cannot be evenly divided by 4, the last list has only one element.

10.3 Exercises

1. Using `cycle`, `split_at`, and `take` you can import from `itertools` and `more_tertools`:

   ```
   from itertools import cycle
   from more_itertools import split_at, take
   ```

   write a one-line solution that:

   a. uses an iterator as input that generates an indefinite stream of strings that looks like this, `'abcdefg...abcdefg...abcdefg...abcdefg....'`

   b. split this stream at the letter c, and

   c. turn the first three elements of the result into a list.

Infinity Unchained

Be careful when working with indefinite datastructures. Materialize only parts of them. Otherwise, you might use up a lot of memory and potentially freeze you computer. You can interrupt a Jupyter Notebook with the key combination `<ESC> + I + I`. This is equivalent to `<CTRL> C` on the command line.

11 Toolz

11.1 A functional library

Toolz is a pure Python package that provides a functional standard library for Python. Installation is simple:

```
pip install toolz
```

This is what Toolz say about itself:

> ***A functional standard library for Python***
>
> Toolz provides a set of utility functions for iterators, functions, and dictionaries. These functions interoperate well and form the building blocks of common data analytic operations. They extend the standard libraries itertools and functools and borrow heavily from the standard libraries of contemporary functional languages.
>
> https://toolz.readthedocs.io/en/latest/

Toolz works only with the basic components iterables, dictionaries, and functions. This makes it simple to understand the used data structures. On the other hand, toolz adds about 60 new functions that need to learned to make good use of toolz.

The documentation explains that toolz provides functions that are:

- *Composable*: They interoperate due to their use of core data structures.
- *Pure*: They don't change their inputs or rely on external state.
- *Lazy*: They don't run until absolutely necessary, allowing them to support large streaming data sets.

There are three main parts of Toolz:

- Itertoolz
- Functoolz
- Dicttoolz

11.2 Currying

Toolz provides support for currying. This function:

```
def add(x, y):
    return x + y
```

can be called with two:

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

but not with only one argument:

```
>>> add(3)
TypeError: add() missing 1 required positional argument: 'y'
```

`curry` from toolz makes currying as simple as applying a decorator:

```
from toolz import curry

@curry
def add(x, y):
    return x + y
```

Now, calling `add()` with one argument:

```
>>> a10 = add(10)
```

returns a new function:

```
>>> a10(7)
17
```

This curried function with three parameters:

```
@curry
def show(a, b, c):
    print(a, b, c)
```

just prints its arguments:

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

Using keyword arguments:

```
>>> sc37 = show(c=37)
```

works as expected:

```
>>> sc37(10, 20)
    10 20 37
```

This allows to bind the first argument:

```
>>> sa20 = show(a=20)
```

Now, b and c cannot be suppled by position any more:

```
>>> sa20(1, 2)
TypeError: show() got multiple values for argument 'a'
```

but need to be keyword arguments:

```
>>> sa20(b=7, c=8)
20 7 8
```

Toolz provides all functions also in a curried version. They are located in `toolz.curried`:

```
>>> from toolz import curried
```

This simple function:

```
L = list(range(2, 11))
def double(x):
    return x * 2
```

can be mapped over all element of an iterable:

```
>>> L = list(range(2, 11))
>>> list(map(double, L))
[4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Here `map()` needs to be called with the function and the iterable. The curried `map()` allows to be called only with a function:

```
>>> double_many = curried.map(double)
```

The resulting function can now be called with only the iterable:

```
>>> list(double_many(L))
[4, 6, 8, 10, 12, 14, 16, 18, 20]
```

11.3 Function Composition

Let's assume we have this three silly functions:

```
def f1(x):
    return x + 2

def f2(x):
    return x + 7

def f3(x):
    return x + 12
```

and apply them in this way:

```
>>> f1(f2(f3(4)))
25
```

This programming style easily leads to an inflation of parenthesis. The function `compose()`:

```
>>> from toolz import compose
```

can help to combine these three functions into one:

```
>>> f_all = compose(f1, f2, f3)
```

Now, we are down from three pairs of parenthesis to one:

```
>>> f_all(4)
25
```

This technique is powerful for combining simple functions into a more complex one. Since a function call is an expression, it can be used anywhere in the code. For this specific case, this is the equivalent by defining a new function with `def`:

```
def f_all_defined(x, f1=f1, f2=f2, f3=f3):
    return f1(f2(f3(x)))
```

```
>>> f_all_defined(4)
25
```

This has some disadvantages:

1. It is a statement and needs to be written on its own lines, while `compose()` can be called anywhere.
2. `f_all_defined` works only for exactly thee functions, whereas `compose()` can take any number of functions.

11.4 Grouping

Grouping is an important task. `toolz.groupby()` allows to group a sequence by a condition expressed as a function. For example, this list of names of some programming languages:

```
languages = ['ALGOL', 'Bash', 'Basic', 'C', 'C++', 'Erlang', 'Fortran', 'Go', 'Haskell',
             'Java', 'Javascript', 'Lisp', 'PHP', 'Python', 'Ruby', 'Rust', 'Swift']
```

can be sorted (by the super useful feature;)) of the length of its name:

```
>>> groupby(len, languages)
{5: ['ALGOL', 'Basic', 'Swift'],
 4: ['Bash', 'Java', 'Lisp', 'Ruby', 'Rust'],
 1: ['C'],
 3: ['C++', 'PHP'],
 6: ['Erlang', 'Python'],
 7: ['Fortran', 'Haskell'],
 2: ['Go'],
 10: ['Javascript']}
```

or can be separated into two groups, (1) shorter than four characters and (2) equal and longer than four characters:

```
>>> short = lambda x: len(x) < 4
>>> groupby(short, languages)
{False: ['ALGOL',
  'Bash',
```

```
  'Basic',
  'Erlang',
  'Fortran',
  'Haskell',
  'Java',
  'Javascript',
  'Lisp',
  'Python',
  'Ruby',
  'Rust',
  'Swift'],
 True: ['C', 'C++', 'Go', 'PHP']}
```

For a list of dictionaries:

```
versions = [
            {'name': 'ALGOL', 'version':  1},
            {'name': 'Bash', 'version':  2},
            {'name': 'Basic', 'version':  3},
            {'name': 'C', 'version':  4},
            {'name': 'C++', 'version':  5},
            {'name': 'Erlang', 'version':  2},
            {'name': 'Fortran', 'version':  90},
            {'name': 'Go', 'version':  1},
            {'name': 'Haskell', 'version':  5},
            {'name': 'Java', 'version':  7},
            {'name': 'Javascript', 'version':  3},
            {'name': 'Lisp', 'version':  7},
            {'name': 'PHP', 'version':  4},
            {'name': 'Python', 'version':  3},
            {'name': 'Ruby', 'version':  2},
            {'name': 'Rust', 'version':  1},
            {'name': 'Swift', 'version':  2},
            ]
```

The grouping can be done by member of the dicts:

```
>>> groupby('version', versions)
{1: [{'name': 'ALGOL', 'version': 1},
  {'name': 'Go', 'version': 1},
  {'name': 'Rust', 'version': 1}],
 2: [{'name': 'Bash', 'version': 2},
  {'name': 'Erlang', 'version': 2},
  {'name': 'Ruby', 'version': 2},
  {'name': 'Swift', 'version': 2}],
 3: [{'name': 'Basic', 'version': 3},
  {'name': 'Javascript', 'version': 3},
  {'name': 'Python', 'version': 3}],
 4: [{'name': 'C', 'version': 4}, {'name': 'PHP', 'version': 4}],
 5: [{'name': 'C++', 'version': 5}, {'name': 'Haskell', 'version': 5}],
 90: [{'name': 'Fortran', 'version': 90}],
 7: [{'name': 'Java', 'version': 7}, {'name': 'Lisp', 'version': 7}]}
```

The function `countby()`:

```
>>> from toolz import countby
```

works in the same way and counts based on return value of a function:

```
>>> countby(len, languages)
{5: 3, 4: 5, 1: 1, 3: 2, 6: 2, 7: 2, 2: 1, 10: 1}
```

or based on a member of a dict:

```
>>> countby('version', versions)
{1: 3, 2: 4, 3: 3, 4: 2, 5: 2, 90: 1, 7: 2}
```

11.5 Exercises

1. Using `dict()` and `zip()`, create a new dictionary:

   ```
   {'a': 10, 'b': 11, 'c': 12, 'd': 13, 'e': 14, 'f': 15, 'g': 16}
   ```

2. Use a dictionary comprehension to create a new dictionary from `d` that dose not contain the keys `c`, `e`, and `f`, i.e.:

   ```
   {'a': 10, 'b': 11, 'd': 13, 'g': 16}
   ```

3. Solve 2. with a function from `toolz`. Hint: Look at Dicttoolz.