Functional Programming in Python

Concepts and Examples

Patrick Bucher

Contents

This overview is inspired by Functional Programming in Python by Martin McBride.

1 Introduction

Python supports three major programming paradigms:

- Procedural Programming: Code is structured in blocks (functions, loops, if statements); simple, but hard to maintain big code bases.

- Object Oriented Programming (OOP): Code is structured in interacting objects; this encapsulation makes independent testing easier, the approach scales better to larger code bases.
- Functional Programmnig (FP): Functions are used as the main building block; a declarative rather than imperative programming style is used.

Those paradigms are usually mixed; however, FP is often neglected.

1.1 Functional Concepts

In FP, functions are *first class objects*: They can be stored in variables, passed to other functions as parameters, or be returned from functions.

Functions that operate on functions are called *higher order functions*.

A *pure function* calculates a result without any side effects. Its result only depends on the input parameters, not on global state. Neither is global state changed. A pure function called multiple times with the same arguments always returns the same result.

FP fits well together with immutable objects such as strings and tuples. Iterators, which do not allow for modification, but for *lazy evaluation*, are often preferred over lists.

Higher order functions (`filter`, `map`, `apply`) and recursion are preferred over structural constructs (`if`/`else` branching, loops).

New functions are created dynamically by combining existing functions.

1.2 Pros and Cons

FP has a lot of *advantages*:

- Conciseness: more can be expressed in less lines of code thanks to more abstract constructs
- Clarity: the programmer's intention is put across better by using higher-order functions like `map` than loop constructs that have to be deciphered line by line
- Provability: without side effects, reasoning about programs is easier; mathematical correctness proofs become *possible*
- Concurrency: without side effects, functions can be executed independently and in parallel and won't cause race conditions

However, this comes with some *disadvantages*:

- Purity is often not possible, because the purpose of many programs is to change global state. A line between pure and impure code has to be drawn.
- A lot of learning effort is required to understand functional concepts such as lambda expressions, closures, partial functions, currying etc.

- Functional code can be less efficient than structured code due to constructs that are less efficient (recursion instead of loops) or more expensive (re-building data structures instead of modifying them).

2 Functions as Objects

Functions can be stored in variables like, say, a string:

```
name = 'Dilbert'

employee = name

def say_hi(name):
    return f'Hello, {name}'

greet = say_hi

print(name)
print(employee)
print(say_hi)
print(greet)
```

There are two variables (*aliases*) pointing to the same object in memory:

```
Dilbert
Dilbert
<function say_hi at 0x7f955db5b040>
<function say_hi at 0x7f955db5b040>
```

It's also possible (but hardly advisable) to overwrite a function reference:

```
def greet(name):
    return f'Hello, {name}'

def say_hi(name):
    return greet(name)

print(say_hi('Dilbert'))

def greet(name):
    return f'Greetings, {name}'

print(say_hi('Wally'))
```

After overwriting the `greet` function, the second implementation is called:

```
Hello, Dilbert
Greetings, Wally
```

The implementation of `say_hi` function has been modified indirectly, which could introduce subtle bugs.

Consider the following conversion functions:

```
def miles_to_kilometers(miles):
    return miles / 1.60934

def usd_to_chf(usd):
    return usd / 0.92

print('500 miles =', miles_to_kilometers(500), 'km')
print('100 usd =', usd_to_chf(100), 'chf')
```

Which perform their conversion independently:

```
500 miles = 310.68636832490336 km
100 usd = 108.69565217391303 chf
```

However, both functions implement the same conversion mechanism, which can be generalized:

```
def convert(f, x):
    return f(x)

print('500 miles =', convert(miles_to_kilometers, 500), 'km')
print('100 usd =', convert(usd_to_chf, 100), 'chf')
```

Both a function and a number are passed to `convert`, which then applies the function to the number. Any conversion can be made, also between different types:

```
def format_currency(x):
    return f'{x:.2f}'

convert(format_currency, 10/3)  # '3.33 chf'
```

2.1 Sorting

The built-in `sorted` function accepts an optional `key` function that allows for customized sorting:

```
dilbert = ('Dilbert', 42)
alice = ('Alice', 37)
dogbert = ('Dogbert', 7)
ashok = ('Ashok', 21)

employees = [dilbert, alice, dogbert, ashok]

def get_name(employee):
    return employee[0]

def get_age(employee):
    return employee[1]

by_name = sorted(employees, key=get_name)
by_age = sorted(employees, key=get_age)

print(by_name)
print(by_age)
```

The list of employees is sorted twice: once by name, and once by age:

```
[('Alice', 37), ('Ashok', 21), ('Dilbert', 42), ('Dogbert', 7)]
[('Dogbert', 7), ('Ashok', 21), ('Alice', 37), ('Dilbert', 42)]
```

2.2 Lambdas

The code can be shortened by using unnamed *lambda functions*:

```
by_name = sorted(employees, key=lambda e: e[0])
by_age = sorted(employees, key=lambda e: e[1])
```

Lambdas consist of a single expression and, thus, should only be used for very simple computations that can be clearly understood without a function name or additional comments. Use a regular function if an expression is used more than once.

Lambdas are function objects that can also be called directly:

```
>>> (lambda x: x ** 2)(5)
24
```

More practically, they can be returned from functions:

```
def create_increment_function(step=1):
    return lambda x: x + step

add_one = create_increment_function()
add_two = create_increment_function(step=2)

add_one(5)  # 5
add_two(5)  # 7
```

Operators are not functions, but can be wrapped in lambda expressions for use with higher-order functions:

```
def calculate(op, a, b):
    return op(a, b)

calculate(lambda a, b: a + b, 3, 1)  # 4
calculate(lambda a, b: a * b, 3, 2)  # 6
```

2.3 Operator Functions

The `operator` module contains pre-defined functions for common operators, so no lambdas have to be implemented:

```
import operator

def calculate(op, a, b):
    return op(a, b)

calculate(operator.add, 3, 1)  # 4
calculate(operator.mul, 3, 2)  # 6
```

See the documentation of the `operator` module for a full list of operators and their function equivalents.

2.4 Partial Function Application

Functions can be *partially applied*, i.e. called with fewer arguments than expected, which returns a function only expecting the missing arguments:

```
from functools import partial

def f(a, b, c, x):
    return a * x**2 + b * x + c

f(2, 4, 6, 1)  # x=1: 2x² + 4x + 6 = 12

g = partial(f, 2, 4, 6)
g(1)   # x=1: 2x² + 4x + 6 = 12
g(2)   # x=2: 2x² + 4x + 6 = 22
```

3 Mutability

Lists, dictionaries, and sets are *mutable*; numbers, strings, and tuples are *immutable*. A frozen set is an immutable version of a set. References to any of those objects are always mutable: by re-assigning a variable, the object pointed to is *not changed*, but *another object* is pointed to instead.

Notice that mutability is shallow. A tuple itself cannot be modified, but the elements of a tuple containing of lists can be modified.

The `sort` method of a list modifies the underlying list, whereas the `sorted` function returns a sorted copy of the given list.

A list can be copied by passing it to the `list` function:

```
def tail(l):
    del l[0]
    return l

numbers = [1, 2, 3]
print(tail(numbers))  # [2, 3]
print(numbers)        # [2, 3], too (modified)

numbers = [1, 2, 3]
print(tail(list(numbers)))  # [2, 3]
print(numbers)              # [1, 2, 3], still (unmodified)
```

This, however, is very inefficient. Instead, the `tail` function could work with slicing to guarantee immutability:

```
def tail(l):
    return l[1:]

numbers = [1, 2, 3]
```

```
print(tail(numbers))  # [2, 3]
print(numbers)        # [1, 2, 3], still (unmodified)
```

Under the hood, slicing is copying, so this solution is not very efficient, too.

Modifications that affect every single item of a list can be expressed using *list comprehensions*:

```
numbers = [1, 2, 3]
twice = [x * 2 for x in numbers]
print(twice)  # [2, 4, 6]
```

4 Recursion

Functions that call themselves are a common technique in functional programming. A problem is thereby reduced to its base case, which is defined statically. In the general case, a problem is simplified towards the base case:

```
def factorial(n):
    if n == 0:
        return 1
    elif n > 0:
        return n * factorial(n - 1)

print(factorial(2))  # 2
print(factorial(3))  # 6
print(factorial(4))  # 24
```

The bigger the argument `n` is chosen, the more functions are running at the same time:

```
factorial(6)
6 * factorial(5)
6 * 5 * factorial(4)
6 * 5 * 4 * factorial(3)
6 * 5 * 4 * 3 * factorial(2)
6 * 5 * 4 * 3 * 2 * factorial(1)
6 * 5 * 4 * 3 * 2 * 1 factorial(0)
6 * 5 * 4 * 3 * 2 * 1 * 1
6 * 5 * 4 * 3 * 2 * 1
6 * 5 * 4 * 3 * 2
6 * 5 * 4 * 6
6 * 5 * 24
6 * 120
720
```

This doesn't scale well. An alternative approach to recursive functions are tail-recursive functions, which carry intermediate results as an extra accumulator parameter (`acc`):

```
def factorial(n, acc=1):
    if n == 0:
        return acc
    elif n > 0:
        return factorial(n-1, n * acc)
```

Which leads to an easier to understand call stack:

```
factorial(6, 1)
factorial(5, 6)
factorial(4, 30)
factorial(3, 120)
factorial(2, 360)
factorial(1, 720)
factorial(0, 720)
720
```

4.1 (No) Tail Call Optimization

Some compilers are able to optimize tail-recursive functions by re-using stack frames for multiple function calls. Unfortunately, Python doesn't support this optimization, so other solutions needs to be considered, such as loops.

Recursion becomes even more inefficient as multiple additional functions are called in each step, as a recursive implementation of a function to compute the Fibonacci numbers requires:

```
def fib(n):
    print(f'fib({n})')
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

The `print` call makes the amount of (redundant) functions being called apparent:

```
>>> fib(6)
fib(6)
fib(4)
fib(2)
```

```
fib(0)
fib(1)
fib(3)
fib(1)
fib(2)
fib(0)
fib(1)
fib(5)
fib(3)
fib(1)
fib(2)
fib(0)
fib(1)
fib(4)
fib(2)
fib(0)
fib(1)
fib(3)
fib(1)
fib(2)
fib(0)
fib(1)
8
```

The `fib` function is called with the argument `1` alone eight times. The inefficiency becomes even more striking when using a bigger `n` and counting the function calls (`fibonacci.py`):

```
calls = 0

def fib(n):
    global calls
    calls += 1
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)

print(f'fib(35)={fib(35)} after {calls} calls')
```

Almost 30 million function calls in a bit less than five seconds are required to compute the 35th Fibonacci number:

```
$ time python3 fibonacci.py
fib(35)=9227465 after 29860703 calls

real    0m4.948s
user    0m4.946s
sys     0m0.000s
```

4.2 Memoization

When many intermediate results are computed multiple times, re-using those results helps saving function calls. For this purpose, the function arguments are (keys) are put together with the results (values) into a dictionary. This technique is called *memoization*:

```
calls = 0
cache = {}

def fib(n):
    global calls
    calls += 1
    if n in cache:
        return  cache[n]
    else:
        if n == 0:
            result = 0
        elif n == 1:
            result = 1
        else:
            result = fib(n-2) + fib(n-1)
        cache[n] = result
        return result

print(f'fib(35)={fib(35)} after {calls} calls')
```

Which reduces function calls by a factor of more than 4*10^5, and runtime by a factor of roughly 167 (memoization comes with a slight overhead).

```
fib(35)=9227465 after 69 calls

real    0m0.030s
user    0m0.030s
sys     0m0.000s
```

Memoization is a *cross cutting concern* that has little to do with the function itself. Python's `functools` has a decorator `lru_cache` (least recently used cache) which provides memoization out-of-the-box:

```
from functools import lru_cache

calls = 0

@lru_cache
def fib(n):
    global calls
    calls += 1
    if n == 0:
        result = 0
    elif n == 1:
        result = 1
    else:
        result = fib(n-2) + fib(n-1)
    return result

print(f'fib(35)={fib(35)} after {calls} calls')
```

Even less functions are invoked, because the caching mechanism is around the function:

```
fib(35)=9227465 after 36 calls

real    0m0.029s
user    0m0.025s
sys     0m0.004s
```

4.3 Flattening Lists

Lists in Python can be nested:

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

Such a list contains numbers and lists, which again contain numbers and lists, and so on. It is often useful to *flatten* such a list:

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

For this purpose, a recursive implementation processes the nested list one by one. The first element (`head`) of the remaining list is considered in each function call, and the remaining elements (`tail`) are delegated to another recursive call. Then, the solution is combined:

```
def flatten(x):
    if not isinstance(x, list):
        # x is a number: first base case
        return [x]
    if x == []:
        # x is an empty list: second base case
        return x
    else:
        # x is a non-empty list: general case
        return flatten(x[0]) + flatten(x[1:])
```

Memoization won't help here, because `x` is different for every function call. A more feasible approach would be to fall back to loops:

```
def flatten(x):
    if not isinstance(x, list):
        # x is a number: first base case
        return [x]
    if x == []:
        # x is an empty list: second base case
        return x
    else:
        # x is a non-empty list: general case
        r = []
        for e in x:
            if isinstance(e, list):
                r += flatten(e)
            else:
                r.append(e)
        return r
```

A recursive function call here only takes place for each additional depth level, not for every additional element.

5 Closures

Functions can contain other functions. The inner function cannnot be seen from the outside of the outer function, unless the outer function returns the inner function. In this example, an inner function `grade` is used from the outer function `grade_exams`.

```
def grade_exams(candidate_scores, max_score):

    def grade(score):
```

```
        # Swiss grades: 1..6
        return score / max_score * 5 + 1

    candidate_grades = {}
    for candidate, score in candidate_scores.items():
        candidate_grades[candidate] = grade(score)

    return candidate_grades

exam_max_score = 50
exam_scores = {
    'Alice': 42,
    'Bob': 35,
    'Mallory': 49,
}
exam_grades = grade_exams(exam_scores, exam_max_score)
print(exam_grades) # {'Alice': 5.2, 'Bob': 4.5, 'Mallory': 5.9}
```

5.1 Returning Inner Functions

Notice how each `score` is passed to `grade`, but `max_score` is taken from the outer scope. The inner function even has access to the outer function's scope if it is returned from the outer function and used elsewhere. The outer function *encloses* the inner function; this construct therefore is called a *closure*:

```
def get_compute_salary_func(year):

    bonus_rates = {
        2018: 0.05,
        2019: 0.10,
        2020: 0.07,
    }
    bonus_rate = bonus_rates.get(year, 0.0)

    def compute_yearly_salary(monthly):
        base_salary = monthly * 12
        bonus = base_salary * bonus_rate
        return base_salary + bonus

    return compute_yearly_salary

compute_2016_salaries = get_compute_salary_func(2016)
compute_2018_salaries = get_compute_salary_func(2018)
```

```
compute_2020_salaries = get_compute_salary_func(2020)

print(compute_2016_salaries(80000))  #  960000.0
print(compute_2018_salaries(80000))  # 1008000.0
print(compute_2020_salaries(80000))  # 1027200.0
```

In the example above, the outer function `get_compute_salary_func` encloses the inner function `compute_yearly_salary`; the latter using the variable `bonus_rate` established in the former's scope. Even though the same function is used multiple times, it computes different results, because the enclosing scope is different.

5.2 Map

A dictionary is a Python data structure that describes the relationship between a key and a value in a static way. The `map` function can be seen as the dynamic counterpart of a `dict`. It is a higher-order function that processes a collection of items using a given function, and returns a collection consisting of the function's return value for each item:

```
max_score = 50
exam_scores = [42, 35, 49]

def grade(score):
    # Swiss grades: 1..6
    return score / max_score * 5 + 1

exam_grades = map(grade, exam_scores)
print(list(exam_grades))  # [5.2, 4.5, 5.9]
```

No explicit looping over the individual scores is needed, the `map` function handles those details. The code can be further simplified by using a lambda instead of a named function:

```
max_score = 50
exam_scores = [42, 35, 49]
exam_grades = map(lambda score: score / max_score * 5 + 1, exam_scores)
print(list(exam_grades))
```

This works with any kind of functions, i.e. also with a closure *primed* with a value, like in the salary example from before:

```
def get_compute_salary_func(year):

    bonus_rates = {
        2018: 0.05,
        2019: 0.10,
        2020: 0.07,
```

```
    }
    bonus_rate = bonus_rates.get(year, 0.0)

    def compute_yearly_salary(monthly):
        base_salary = monthly * 12
        bonus = base_salary * bonus_rate
        return base_salary + bonus

    return compute_yearly_salary

compute_2020_salaries = get_compute_salary_func(2020)
base_salaries = [80000, 90000, 100000]
total_salaries = map(compute_2020_salaries, base_salaries)
print(list(total_salaries))
```

5.3 Composing Functions

Consider the value `x` that has to be processed by two functions `f` and `g`:

1. `y` is computed as `y=g(x)` (intermediate result)
2. `z` is computed as `z=f(y)` (final result)

This, of course, can be simplified by *composing* the two functions `f` and `g` as `f(g(x))`.

Consider this example, where exam scores are first mapped to exam grades, which then are rounded in a second step:

```
def get_grade_for_func(max_score):

    def grade(score):
        return score / max_score * 5 + 1

    return grade

def get_round_to_func(granularity):

    def round_to(value):
        scaled_up = value * (1 / granularity)
        rounded = round(scaled_up)
        scaled_down = rounded * granularity
        return scaled_down

    return round_to
```

```
max_score = 72
scores = [46, 70, 53, 38, 67]

grade_for = get_grade_for_func(max_score)
round_to = get_round_to_func(0.1)

exact_grades = map(grade_for, scores)
rounded_grades = map(round_to, exact_grades)

print(list(rounded_grades))  # [4.2, 5.9, 4.7, 3.6, 5.7]
```

This approach requires two calls to `map`, with each call iterating over all the elements. If the grader and rounding function are composed to a single function, the list only needs to be processed once:

```
def get_grade_for_func(max_score):

    def grade(score):
        return score / max_score * 5 + 1

    return grade

def get_round_to_func(granularity):

    def round_to(value):
        scaled_up = value * (1 / granularity)
        rounded = round(scaled_up)
        scaled_down = rounded * granularity
        return scaled_down

    return round_to

max_score = 72
scores = [46, 70, 53, 38, 67]

grade_for = get_grade_for_func(max_score)
round_to = get_round_to_func(0.1)

def compose(f, g):

    def func(x):
        return f(g(x))

    return func
```

```
score_to_rounded_grade = compose(round_to, grade_for)

rounded_grades = map(score_to_rounded_grade, scores)

print(list(rounded_grades))  # [4.2, 5.9, 4.7, 3.6, 5.7]
```

This approach scales much better: Not only in terms of runtime efficiency, which becomes noticable as the number of elements grows, but only if additional computations need to be done for every item.

In this example, an additional point bonus is added to each score, so that the maximum grade can be reached without a perfect score:

```
def get_bonus_of_func(bonus):

    def add(score):
        return score + bonus

    return add

def get_grade_for_func(max_score):

    def grade(score):
        return score / max_score * 5 + 1

    return grade

def get_round_to_func(granularity):

    def round_to(value):
        scaled_up = value * (1 / granularity)
        rounded = round(scaled_up)
        scaled_down = rounded * granularity
        return scaled_down

    return round_to

max_score = 100
scores = [70, 80, 90, 40, 100]

bonus_of = get_bonus_of_func(max_score / 10)
grade_for = get_grade_for_func(max_score)
round_to = get_round_to_func(0.1)
```

```
def compose(f, g):

    def func(x):
        return f(g(x))

    return func

score_to_exact_grade = compose(grade_for, bonus_of)
score_to_rounded_grade = compose(round_to, score_to_exact_grade)

rounded_grades = map(score_to_rounded_grade, scores)

print(list(rounded_grades))  # [5.0, 5.5, 6.0, 3.5, 6.5]
```

Unfortunately, this brings up another issues: Grades higher than the maximum grade of 6.0 are computed. However, this issues can be solved by composing even further:

```
def get_bonus_of_func(bonus):

    def add(score):
        return score + bonus

    return add

def get_grade_for_func(max_score):

    def grade(score):
        return score / max_score * 5 + 1

    return grade

def get_limit_of_func(max_grade):

    def limit(grade):
        return min(grade, max_grade)

    return limit


def get_round_to_func(granularity):

    def round_to(value):
        scaled_up = value * (1 / granularity)
        rounded = round(scaled_up)
```

```
        scaled_down = rounded * granularity
        return scaled_down

    return round_to

max_score = 100
scores = [70, 80, 90, 40, 100]

bonus_of = get_bonus_of_func(max_score / 10)
grade_for = get_grade_for_func(max_score)
limit_of = get_limit_of_func(6.0)
round_to = get_round_to_func(0.1)

def compose(f, g):

    def func(x):
        return f(g(x))

    return func

score_to_exact_grade = compose(grade_for, bonus_of)
score_to_bounded_grade = compose(limit_of, score_to_exact_grade)
score_to_rounded_grade = compose(round_to, score_to_bounded_grade)

rounded_grades = map(score_to_rounded_grade, scores)

print(list(rounded_grades))  # [5.0, 5.5, 6.0, 3.5, 6.0]
```

Compare this to a procedural approach, which is much shorter in terms of lines:

```
max_score = 100
scores = [70, 80, 90, 40, 100]
bonus = max_score / 10
max_grade = 6.0
granularity = 0.1

grades = []
for score in scores:
    score = score + bonus
    grade = score / max_score * 5 + 1
    if grade > max_grade:
        grade = max_grade
    grade = round(grade * 1 / granularity) * granularity
    grades.append(grade)
```

```
print(grades)  # [5.0, 5.5, 6.0, 3.5, 6.0]
```

However, this code is harder to reason about ("Where did the error happen?"), especially if the computations are getting more involved. The functional approach allows you to reason about and write tests for each function in isolation. If the functions work correctly, are composed in the right way and used with well-tested higher-order functions like `map` , the result will be correct, too.

5.4 Closures vs. Classes

Like objects, closures can hold state. In OOP, the state can be initialized using a constructor. A method of the same class then can perform computations based on both internal state and parameters:

```
class Rounder:

    def __init__(self, granularity):
        self._granularity = granularity

    def round(self, value):
        scaled_up = value * (1 / self._granularity)
        rounded = round(scaled_up)
        scaled_down = rounded * self._granularity
        return scaled_down

grades = [5.234, 4.738, 3.269]
rounder = Rounder(0.05)
rounded = map(rounder.round, grades)
print(list(rounded))  # [5.25, 4.75, 3.25]
```

Python has a special method `__call__`, which allows objects to be used like functions. The above implementation can be turned more pythonesque by, first, renaming `round` to `__call__`, and, second, by using `rounder` as a function (instead of its method `rounder.round`). Calls to `rounder()` will be delegated to the `__call__` method:

```
class Rounder:

    def __init__(self, granularity):
        self._granularity = granularity

    def __call__(self, value):
        scaled_up = value * (1 / self._granularity)
        rounded = round(scaled_up)
```

```
        scaled_down = rounded * self._granularity
        return scaled_down

grades = [5.234, 4.738, 3.269]
rounder = Rounder(0.05)
rounded = map(rounder, grades)
print(list(rounded))  # [5.25, 4.75, 3.25]
```

This approach is useful when objects first need to be configured in a complicated but inconsistent manner. Think of the Builder pattern, that allows to initialize objects only using a subset of available parameters:

```
class Salary:

    _bonus = 0
    _taxes = 0
    _penalty = 0

    def __init__(self, amount):
        self._salary = amount

    def with_bonus(self, rate):
        self._bonus = rate
        return self

    def with_taxes(self, rate):
        self._taxes = rate
        return self

    def with_penalty(self, penalty):
        self._penalty = penalty
        return self

    def __call__(self):
        pre_bonus = (self._salary - self._penalty)
        pre_taxes = pre_bonus + pre_bonus * self._bonus
        return pre_taxes - pre_taxes * self._taxes

salary_1 = Salary(100000).with_bonus(0.1).with_penalty(5000)
salary_2 = Salary(100000).with_bonus(0.1).with_taxes(0.2)
print(salary_1())  # 104500.0
print(salary_2())  #  88000.0
```

A function returning a closure requires optional parameters for the same purpose:

```
def get_salary_func(bonus=0, taxes=0, penalty=0):

    def compute(salary):
        pre_bonus = (salary - penalty)
        pre_taxes = pre_bonus + pre_bonus * bonus
        return pre_taxes - pre_taxes * taxes

    return compute

salary_1 = get_salary_func(bonus=0.1, penalty=5000)
salary_2 = get_salary_func(bonus=0.1, taxes=0.2)
print(salary_1(100000))  # 104500.0
print(salary_2(100000))  #  88000.0
```

5.5 Inspecting Closures

Python provides the special attributes `__closure__` and `__code__` to inspect closures (see the Data Model for details).

The variables a function has access to by enclosing an outer scope—so-called *free variables*—can be retrieved as a tuple using the `co_freewars` attribute of the `__code__` attribute. To get the values of those free variables, inspect the `cell_contents` attribute of each element of the `__closure__` attribute:

```
def get_salary_func(bonus=0, taxes=0, penalty=0):

    def compute(salary):
        pre_bonus = (salary - penalty)
        pre_taxes = pre_bonus + pre_bonus * bonus
        return pre_taxes - pre_taxes * taxes

    return compute

salary = get_salary_func(bonus=0.1, penalty=5000)
print(salary.__code__.co_freevars)  # ('bonus', 'penalty', 'taxes')
print(salary.__closure__[0].cell_contents)  # 0.1
print(salary.__closure__[1].cell_contents)  # 5000
print(salary.__closure__[2].cell_contents)  # 0
```

This process can be simplified using an utility function:

```
def get_salary_func(bonus=0, taxes=0, penalty=0):

    def compute(salary):
```

```
        pre_bonus = (salary - penalty)
        pre_taxes = pre_bonus + pre_bonus * bonus
        return pre_taxes - pre_taxes * taxes

    return compute

salary = get_salary_func(bonus=0.1, penalty=5000)

def inspect_closure(func):
    for i, name in enumerate(func.__code__.co_freevars):
        print(f'{name} = {func.__closure__[i].cell_contents}')

inspect_closure(salary)
```

Which outputs all the free variables of a closure:

```
bonus = 0.1
penalty = 5000
taxes = 0
```

Notice that those are read-only values, don't attempt to manipulate those closures: better create a new one.

6 Iterators

An *iterator* can be used to process the elements of a sequence one by one. When passed to the `next()` function, the next element of the iterator's underlying sequence is returned—or `StopIteration` thrown, in case the iterator is *exhausted*, i.e. all of its elements have been processed.

Higher-order functions like `filter` or `map` return iterators:

```
numbers = [1, 2, 3, 4, 5]
even = filter(lambda x: x % 2 == 0, numbers)
odd = map(lambda x: x + 1, even)
print(next(odd))  # 3
print(next(odd))  # 5
print(next(odd))  # StopIteration
```

An iterator can onle be processed once in forward direction. However, multiple iterators can be used to process the same underlying sequence.

6.1 Iterables

An *iterable* is something (usually a sequence like list, tuple, string) that can be turned into an iterator by passing it to the `iter()` function, which returns a new iterator:

```
numbers = [1, 2, 3]
i = iter(numbers)
print(next(i))  # 1
print(next(i))  # 2
print(next(i))  # 3
```

An iterator itself is also an iterable, so calls to `iter()` passing an iterator return the same iterator with its current state:

```
numbers = [1, 2, 3]
i = iter(numbers)
print(next(i))  # 1
j = iter(i)
print(next(j))  # 2
print(next(i))  # 3
```

6.2 Loops use Iterators

Internally, Python relies heavily on iterators. A `for/in` loop works on any iterable. First, `iter()` is called on the loop's iterable to get an iterator. Then, for every iteration, `next()` is called on the iterator to get to the next elements. Finally, the loop ends when `StopIteration` is raised.

Consider this `for/in` loop:

```
numbers = [1, 2, 3]

for x in numbers:
    print(x)
```

Which could be re-written using explicit `iter()` and `next()` calls and a `while` loop:

```
numbers = [1, 2, 3]

i = iter(numbers)
while True:
    try:
        x = next(i)
        print(x)
```

```
    except StopIteration:
        break
```

6.3 Lazy Evaluation

Iterators only must produce their values when requested using the `next()` function, which means that they can use *lazy evaluation*. If an iteration is stopped before the iterator has been exhausted, no remaining items have been computed in vain. This can save computing power and memory, but potentially increases the processing time needed for a single iteration. (Picking between lazy and eager evaluation is a trade-off.) Iterators implemented using lazy evaluation can be of infinite length.

The built-in `range()` function produces lazy sequences. However, the sequence's length can be figured out using the built-in `len()` function considering the limit arguments given to `range()`: `len(range(1, 5))` is `5 - 1 = 4`.

6.4 Realizing Iterators

An iterator must be *realized* before all of its items can be dealt with at once, i.e. by printing out the whole sequence of items. For this purpose, the according functions can be called:

```
numbers = range(1, 4)

print(list(iter(numbers)))  # [1, 2, 3]
print(set(iter(numbers)))   # {1, 2, 3}
print(tuple(iter(numbers))) # (1, 2, 3)
```

Alternatively, the expansion operator `*` can be used with according literals:

```
numbers = range(1, 4)

print([*iter(numbers)])  # [1, 2, 3]
print({*iter(numbers)})  # {1, 2, 3}
print((*iter(numbers),)) # (1, 2, 3)
```

Notice the trailing comma required for tuple expansion in the last example.

When working with strings, `str` will call the iterator's implementation of the `__str__()` dunder method, which describes the iterator itself rather than its items. Use the `join()` method on an empty string to realize a list of characters:

```
offsets = range(0, 26)
capital_a = 65
alphabet = map(lambda c: chr(c + capital_a), offsets)

print(str(alphabet))      # <map object at 0x7fa5d875b4f0>
print(''.join(alphabet)) # ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

6.5 Implementing an Iterator

An iterator can be implemented by providing two dunder methods: `__next__()` and `__iter__()`. Calls of the built-in functions `next()` and `iter()` will be forwarded to the argument's respective dunder methods.

The class `Factorials` implements an iterator that provides the successive factorial numbers up to a limit passed to the constructor. The implementation uses lazy evaluation:

```
class Factorials():

    def __init__(self, n):
        if n < 0:
            raise ValueError('n! is only defined for n >= 0')
        self.n = n
        self.i = 0
        self.x = 1

    def __iter__(self):
        return self

    def __next__(self):
        if self.i < self.n:
            self.i += 1
            self.x *= self.i
            return self.x
        else:
            raise StopIteration


print(list(Factorials(3))) # [1, 2, 6]
print(list(Factorials(5))) # [1, 2, 6, 24, 120]
print(list(Factorials(8))) # [1, 2, 6, 24, 120, 720, 5040, 40320]
```

In practice, *generators* are often a better fit for such tasks.

7 Transforming Iterables

Python provides functions to transform iterables, which are less prone to side effects, and therefore the better fit than lists from a functional perspective.

7.1 Enumerating

The built-in `enumerate()` function transforms a sequence into an iterator of tuples, each containing an index value and an item from the original sequence:

```
names = ['Alice', 'Bob', 'Mallory']
for item in enumerate(names):
    print(item)
```

```
(0, 'Alice')
(1, 'Bob')
(2, 'Mallory')
```

An optional start index can be provided, and the tuple can be unpacked using two variables for the loop:

```
names = ['Alice', 'Bob', 'Mallory']
for index, name in enumerate(names, 1):
    print(index, name)
```

```
1 Alice
2 Bob
3 Mallory
```

7.2 Zipping and Unzipping

Multiple sequences can be processed together using the built-in `zip()` function:

```
names = ['Dilbert', 'Dogbert', 'Ashok']
jobs = ['Engineer', 'Consultant', 'Intern']
salaries = [120000, 250000, 18000]

for employee in zip(names, jobs, salaries):
    print(employee)
```

```
('Dilbert', 'Engineer', 120000)
('Dogbert', 'Consultant', 250000)
('Ashok', 'Intern', 18000)
```

Again, the tuple can be unpacked by using multiple variables for the loop:

```
names = ['Dilbert', 'Dogbert', 'Ashok']
jobs = ['Engineer', 'Consultant', 'Intern']
salaries = [120000, 250000, 18000]

for name, job, salary in zip(names, jobs, salaries):
    print(name, job, salary)
```

```
Dilbert Engineer 120000
Dogbert Consultant 250000
Ashok Intern 18000
```

Notice that `zip()` stops when the shortest sequence is exhausted:

```
names = ['Dilbert', 'Dogbert', 'Ashok']
jobs = ['Engineer']
salaries = [120000, 250000]

for employee in zip(names, jobs, salaries):
    print(employee)
```

```
('Dilbert', 'Engineer', 120000)
```

If the original sequences (`names`, `jobs`, `salaries`) are considered columns of an employee database, the results of the `zip()` operation can be seen as its rows. This transformation can be reversed using `zip()`—by first unpacking the resulting sequence, and then zipping it:

```
names = ['Dilbert', 'Dogbert', 'Ashok']
jobs = ['Engineer', 'Consultant', 'Intern']
salaries = [120000, 250000, 18000]

employees = zip(names, jobs, salaries)
for col in zip(*employees):
    print(col)
```

```
('Dilbert', 'Dogbert', 'Ashok')
('Engineer', 'Consultant', 'Intern')
(120000, 250000, 18000)
```

7.3 Sorting and Reversing

Unlike the list's `sort()` method that sorts a list in-place, the built-in `sorted()` function returns a sorted new list. Either operation allows for an optional `key` argument, which defines the sorting criterion in terms of a function applied to every item:

```
names = ['Dilbert', 'Dogbert', 'Ashok']
jobs = ['Engineer', 'Consultant', 'Intern']
salaries = [120000, 250000, 18000]
employees = zip(names, jobs, salaries)

for employee in sorted(employees, key=lambda e: e[2]):
    print(employee)
```

```
('Ashok', 'Intern', 18000)
('Dilbert', 'Engineer', 120000)
('Dogbert', 'Consultant', 250000)
```

The lambda accessing the tuple element at index 2 can also be taken from the `operator` module, which provides an `itemgetter` function that produces a closure to access the right element:

```
from operator import itemgetter

names = ['Dilbert', 'Dogbert', 'Ashok']
jobs = ['Engineer', 'Consultant', 'Intern']
salaries = [120000, 250000, 18000]
employees = zip(names, jobs, salaries)

for employee in sorted(employees, key=itemgetter(2)):
    print(employee)
```

```
('Ashok', 'Intern', 18000)
('Dilbert', 'Engineer', 120000)
('Dogbert', 'Consultant', 250000)
```

When dealing with classes instead of tuple, use the `attrgetter` function to access attributes by name. The `methodcaller` function allows to call any method on each item by its name:

```
from operator import methodcaller

names = ['POINTY HAIRED BOSS', 'Dilbert', 'dogbert', 'alice']
for name in sorted(names, key=methodcaller('lower')):
    print(name)
```

Here, the `lower()` method is called on every name in order to sort the names in a case-insensitive manner.

The sort order can be reversed either by setting the optional `reverse` argument of the `sorted()` function to `True`, or by calling the `reversed()` built-in function:

```
names = ['Dilbert', 'Alice', 'Pointy Haired Boss', 'Dogbert', 'Ted']

names_desc = sorted(names, reverse=True)
print(names_desc)

names_desc = reversed(sorted(names))
print(list(names_desc))


['Ted', 'Pointy Haired Boss', 'Dogbert', 'Dilbert', 'Alice']
['Ted', 'Pointy Haired Boss', 'Dogbert', 'Dilbert', 'Alice']
```

Notice that the return value of `reversed()` needs to be realized first.

Function	accepts	returns
`sorted()`	iterable	list
`reversed()`	sequence	iterator

The sorting operations are *stable*, so sorting multiple times will always return the same order of items that share the same sorting criterion, but differ otherwise:

```
# Swiss-German date format
dates = [
    '24.06.1987',
    '13.05.1987',
    '31.12.1988',
    '31.07.1987',
    '17.09.1988',
    '05.02.1987',
    '01.03.1988',
]

by_year_1 = sorted(dates, key=lambda d: d[6:])
by_year_2 = sorted(by_year_1, key=lambda d: d[6:])
by_year_3 = sorted(by_year_2, key=lambda d: d[6:])
for date_1, date_2, date_3 in zip(by_year_1, by_year_2, by_year_3):
    print(date_1, '==', date_2, '==', date_3)
```

```
24.06.1987 == 24.06.1987 == 24.06.1987
13.05.1987 == 13.05.1987 == 13.05.1987
31.07.1987 == 31.07.1987 == 31.07.1987
05.02.1987 == 05.02.1987 == 05.02.1987
31.12.1988 == 31.12.1988 == 31.12.1988
17.09.1988 == 17.09.1988 == 17.09.1988
01.03.1988 == 01.03.1988 == 01.03.1988
```

When counting backwards, the `reverse()` function can be used to create more readable code:

```
range_reverse = range(9, -1, -1)     # hard to read
reversed_range = reversed(range(10)) # easy to read

for a, b in zip(range_reverse, reversed_range):
    print(a, b)
```

```
9 9
8 8
7 7
6 6
5 5
4 4
3 3
2 2
1 1
0 0
```

7.4 Pipelines

Adding `print()` calls to functions used with `filter()` and `map()` shows in which order those functions are executed:

```
def is_taxable(salary):
    print(f'is_taxable({salary})')
    return salary > 100000

def calc_tax(salary):
    print(f'calc_tax({salary})')
    return salary * 0.05

salaries = [120000, 84000, 52000, 190000]
taxable = filter(is_taxable, salaries)
```

```
taxes = map(calc_tax, taxable)

for tax in taxes:
    print(tax)


is_taxable(120000)
calc_tax(120000)
6000.0
is_taxable(84000)
is_taxable(52000)
is_taxable(190000)
calc_tax(190000)
9500.0
```

Notice that those items are processed in a *pipeline* one by one. Even though the call to `map()` comes after the call to `filter()`, the `is_taxable()` operation used by `filter()` has only been processed for the first element yet!

Removing the `taxable` intermediary variable and calling `map()` directly on the result of `filter()` therefore won't have any impact on the order of processing:

```
def is_taxable(salary):
    print(f'is_taxable({salary})')
    return salary > 100000

def calc_tax(salary):
    print(f'calc_tax({salary})')
    return salary * 0.05

salaries = [120000, 84000, 52000, 190000]
taxes = map(calc_tax, filter(is_taxable, salaries))

for tax in taxes:
    print(tax)


is_taxable(120000)
calc_tax(120000)
6000.0
is_taxable(84000)
is_taxable(52000)
is_taxable(190000)
calc_tax(190000)
9500.0
```

But leave the loop at the bottom away, and *no items will be processed at all*:

```
def is_taxable(salary):
    print(f'is_taxable({salary})')
    return salary > 100000

def calc_tax(salary):
    print(f'calc_tax({salary})')
    return salary * 0.05

salaries = [120000, 84000, 52000, 190000]
taxes = map(calc_tax, filter(is_taxable, salaries))
print(taxes)
```

```
<map object at 0x7fd3bbf03fd0>
```

This demonstrates that `filter()`, `map()` an the like use *lazy evaluation*.

7.5 Multiple Map Parameters

The `map()` function can be used on multiple sequences in one go—if used with a function that expects the same number of arguments as sequences are used:

```
numbers = [7, 4, 3, 2]
factors = [1.0, 1.5, 0.5, 2.0]

results = map(lambda n, f: n * f, numbers, factors)
print(list(results)) # [7.0, 6.0, 1.5, 4.0]
```

Again, instead of defining a lambda, an `operator` can be used:

```
from operator import mul

numbers = [7, 4, 3, 2]
factors = [1.0, 1.5, 0.5, 2.0]

results = map(mul, numbers, factors)
print(list(results)) # [7.0, 6.0, 1.5, 4.0]
```

Any number of sequences can be passed to `map()`, as long as the operation performed on them accepts the same number of parameters:

```
def f(a, b, x):
    y = a * x + b
    return y

slopes = [1, 2, 3, 4]
coefficients = [1, 0, 2, 0]
xs = [1.5, 3.0, 2.5, 0.0]

results = map(f, slopes, coefficients, xs)
print(list(results)) # [2.5, 6.0, 9.5, 0.0]
```

8 Reducing Iterables

A *reducing* function combines all the values of an iterable and produces a single value out of them as a result.

8.1 Built-in Reducing Functions

the `len()` function is one of the most common reducing functions. It returns the number of elements contained in an sequence:

```
print(len([7, 3, 5, 2])) # 4
print(len([]))           # 0
```

The `sum()` function adds up the items of an iterable and returns their sum:

```
print(sum([7, 3, 5, 2])) # 17
print(sum([]))           #  0
```

An optional start value can be provided for the second argument if the summation should start from a different value than 0 (default):

```
print(sum([7, 3, 5, 2], -10)) # 7
print(sum([], 0))             # 0
```

Even though the `sum()` function applies the + operator to the elements of the given iterable, it cannot be used to concatenate strings. Use the string's `join()` method instead:

```
letters = ['abc', 'de', 'f', 'ghi']
print(sum(letters, '')) # wrong: TypeError
print(''.join(letters)) # right: abcdefghi
```

The `min()` and `max()` function return the smallest or biggest element of an iterable, respectively:

```
numbers = [7, 3, 1, 9, 5]
print(min(numbers)) # 1
print(max(numbers)) # 9
```

If the elements are list themselves, those sub-lists are compared element-wise:

```
numbers = [[3, 1, 2], [9, 1, 3], [1, 9, 8]]
print(min(numbers)) # [1, 9, 8]
print(max(numbers)) # [9, 1, 3]
```

Calling `min()` or `max()` on an empty iterable causes a `ValueError`, which can be prevented by setting a `default` argument, which is used as a fallback, and ignored for non-empty iterables:

```
numbers = [9, 1, 5]
nothing = []

print(min(numbers))            # 1
print(min(numbers, default=0)) # 1
print(min(nothing))            # ValueError
print(min(nothing, default=0)) # 0

print(max(numbers))            # 9
print(max(numbers, default=0)) # 9
print(max(nothing))            # ValueError
print(max(nothing, default=0)) # 0
```

The optional `key` argument can be used to speficy the criterion being used for comparison—like for the `sorted()` function or `sort()` method:

```
employees = [
    ('Dilbert', 42, 120000),
    ('Alice', 39, 110000),
    ('Wally', 53, 130000),
    ('Ashok', 23, 36000),
]

youngest = min(employees, key=lambda e: e[1])
oldest = max(employees, key=lambda e: e[1])

lowest_salary = min(employees, key=lambda e: e[2])
highest_salary = max(employees, key=lambda e: e[2])

print(f'age: {youngest} (youngest), {oldest} (oldest)')
print(f'earns: {lowest_salary} (least), {highest_salary} (most)')
```

```
age: ('Ashok', 23, 36000) (youngest), ('Wally', 53, 130000) (oldest)
earns: ('Ashok', 23, 36000) (least), ('Wally', 53, 130000) (most)
```

The `any()` function returns `True` if *at least one element* of the given iterable evaluates to `True`:

```
print(any([False, False, False])) # False
print(any([False, False, True]))  # True
print(any([0, 0, 0, 0, 0]))       # False
print(any([0, 0, 0, 1, 0]))       # True
print(any(['', '', '', '']))      # False
print(any(['', 'x', '', 'y']))    # True
print(any([False, '', 0, []]))    # False
print(any([]))                    # False
```

For an empty iterable (last example), `any()` returns False—unlike the `all()` function, which returns `True` if *all elements* evaluate to `True`, and `False`, if an element evaluates to `False`:

```
print(all([True, False, True]))  # False
print(all([True, True, True]))   # True
print(all([2, 8, 0, 3, 8]))      # False
print(all([2, 8, 4, 3, 8]))      # True
print(all(['a', 'b', '', 'd']))  # False
print(all(['u', 'v', 'x', 'y'])) # True
print(all([True, 'a', 1, []]))   # False
print(all([]))                   # True
```

8.2 The `reduce()` Function

The `functools` module provides a `reduce()` function, which allows for custom definitions of reducing operations. Its first argument is a function accepting *two* parameters (the elements `n-1` and `n` to be combined), and its second argument is the iterable to be reduced. This example implements factorials using the `operator` module's `mul()` and the `functool` module's `reduce()` function:

```
from functools import reduce
from operator import mul

def factorial(x):
    numbers = range(1, x+1)
    return reduce(mul, numbers)

print(factorial(4)) #  24
```

```
print(factorial(5)) # 120
print(factorial(6)) # 720
```

As an optional third argument, an `initializer` can be provided:

```
from functools import reduce
from operator import mul

numbers = range(1, 6)
half_the_fac = reduce(mul, numbers, 0.5)
print(half_the_fac) # 60.0
```

This is especially useful when dealing with empty iterables, which result in a `TypeError` when reduced without an initializer, which serves as a fallback value:

```
from functools import reduce
from operator import mul

print(reduce(mul, []))      # TypeError
print(reduce(mul, [], 0.5)) # 0.5
```

8.3 The `filter()`, `map()`, `reduce()` Pattern

Even though they work completely different, the functions `filter()`, `map()`, and `reduce()` have some pair-wise commonalities:

- Both `filter()` and `map()` process the elements of an iterable one by one.
- Both `map()` and `reduce()` transform values.
- Both `filter()` and `reduce()` decrease the number of elements.

Those three functions are often used together to process iterables, resulting in a single value. Consider the following list containing employees, their hourly rates, and the amount of hours worked by each for a project:

```
efforts = [
    # (name, rate, hours)
    ('Dilbert', 220, 13.5),
    ('Alice', 180, 16.0),
    ('Wally', 150, 0.0),
    ('Ashok', 80, 42.5),
    ('Dogbert', 250, 3.5),
    ('Pointy Haired Boss', 500, 0.0),
]
```

In order to produce the total labor costs of the project, this list of tuples can be processed in three steps:

1. *filter*: Only entries with actual working hours (> 0.0) are retained.
2. *map*: Compute the cost for each employee (rate multiplied by hours).
3. *reduce*: Sum up all the individual costs of each employee.

```
from functools import reduce
from operator import add

efforts = [
    # (name, rate, hours)
    ('Dilbert', 220, 13.5),
    # ...
]

involved = filter(lambda e: e[2] > 0.0, efforts)
cost_per_employee = map(lambda e: e[1] * e[2], involved)
total_costs = reduce(add, cost_per_employee)
print(f'total costs: {total_costs}') # 10125.0
```

In this particular example, the `filter` step is redundant, because employees with zero hours would not affect the total cost at all. The `reduce` step could also be simplified using the `sum()` function:

```
efforts = [
    # (name, rate, hours)
    ('Dilbert', 220, 13.5),
    # ...
]

cost_per_employee = map(lambda e: e[1] * e[2], efforts)
total_costs = sum(cost_per_employee)
print(f'total costs: {total_costs}') # 10125.0
```

Consider another example: a list of exam submissions consisting of the name, the submission date, and the score achieved:

```
submissions = [
    # name, submission date, score
    ('Alice', '2021-07-03', 73),
    ('Bob', '2021-07-18', 81),
    ('Charles', '2021-07-12', 57),
    ('Deborah', '2021-07-10', 96),
    ('Ernest', '2021-07-19', 89),
```

```
    ('Fanny', '2021-07-06', 61),
]
```

The average grade of submissions within deadline should be computed as follows:

1. *filter*: Submissions after the deadline (`2021-07-10`) are ignored.
2. *map*: A grade from 1 (worst) to 6 (best) is computed based on a maximum score of 100.
3. *reduce*: The grade average of all submissions is calculated.

```
from datetime import datetime

submissions = [
    # name, submission date, score
    ('Alice', '2021-07-03', 73),
    # ...
]

max_score = 100

def is_within_deadline(submission):
    deadline = datetime.fromisoformat('2021-07-10')
    submitted = datetime.fromisoformat(submission[1])
    return submitted < deadline

def swiss_grade(score, max_score):
    return score / max_score * 5 + 1

within_deadline = filter(is_within_deadline, submissions)
grades = map(lambda s: swiss_grade(s[2], max_score), within_deadline)
grades = list(grades)
average = sum(grades) / len(grades)
print(f'average: {average}') # 4.35
```

9 Comprehensions

Creating an iterable based on another iterable, say, building the squares of a list of numbers, can be done in various ways.

The structured approach uses a `for` loop:

```
numbers = range(1, 10)

squares = []
for number in numbers:
```

```
    squares.append(number ** 2)

print(squares) # [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

This approach is perfectly valid, but requires *operational reasoning* to understand.

A more declarative approach uses the higher-order `map` function, which requires less code to be written:

```
numbers = range(1, 10)

squares = list(map(lambda x: x ** 2, numbers))

print(squares) # [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

However, the best tool for this purpose—building a list based on an iterable—is a list comprehension:

```
numbers = range(1, 10)

squares = [x ** 2 for x in numbers]

print(squares) # [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

No lambda expression is required, the expression can be stated directly.

The comprehension has the following structure:

```
[{expression} for {item} in {iterable}]
```

The above example can be read in English as:

> make a list of `x ** 2` for all values of `x` in `numbers`

9.1 Conditions

The higher-order functinos `filter` and `map` are often used toghether: first, the items to be processed are fitered, second, the remaining items are mapped.

Consider this example turning a list of empty and non-empty strings into title-cased strings, ignoring the empty ones:

```
strings = ['', '', 'john', '', 'alice', '', 'bob']
non_empty = filter(len, strings)
names = list(map(lambda s: s.title(), non_empty))
print(names) # ['John', 'Alice', 'Bob']
```

A comprehension has an optional `if` statements; only items passing this test end up in the resulting sequence:

```
strings = ['', '', 'john', '', 'alice', '', 'bob']
names = [s.title() for s in strings if s]
print(names) # ['John', 'Alice', 'Bob']
```

This code is shorter and clearer. Consider a comprehension as an alternative of combining `filter` and `map`.

9.2 Nesting

Comprehensions can be nested, which can be used to create multi-dimensional lists:

```
def field_2d(rows, cols):
    return [[(x, y) for x in range(cols)] for y in range(rows)]

field = field_2d(6, 7)
for row in field:
    print(row)
```

```
[(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0)]
[(0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1)]
[(0, 2), (1, 2), (2, 2), (3, 2), (4, 2), (5, 2), (6, 2)]
[(0, 3), (1, 3), (2, 3), (3, 3), (4, 3), (5, 3), (6, 3)]
[(0, 4), (1, 4), (2, 4), (3, 4), (4, 4), (5, 4), (6, 4)]
[(0, 5), (1, 5), (2, 5), (3, 5), (4, 5), (5, 5), (6, 5)]
```

Again, this is much shorter than using the structured approach:

```
def field_2d(rows, cols):
    field = []
    for y in range(rows):
        row = []
        for x in range(cols):
            row.append((x, y))
        field.append(row)
    return field
```

Notice that comprehensions can be nested without creating multi-dimensional sequences as a result:

```
coords = [x + y for x in range(0, 40, 10) for y in range(4)]
print(coords)
```

```
[0, 1, 2, 3, 10, 11, 12, 13, 20, 21, 22, 23, 30, 31, 32, 33]
```

This translates to structured code as follows:

```
coords = []
for x in range(0, 40, 10):
    for y in range(4):
        coords.append(x + y)
```

In the comprehension expression, the inner loop is on the right, the outer loop on the left.

9.3 Dictionaries, Sets, Tuples

Comprehensions can be used for the other sequence types—dictionaries, sets, and tuples—too:

```
squares = {x: x ** 2 for x in range(1, 6)}
print(squares) # {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

additions = [(3, 4), (4, 3), (5, 2), (3, 1), (4, 2)]
sums = {x + y for (x, y) in additions}
print(sums) # {4, 6, 7}

strings = ['', '', 'john', '', 'alice', '', 'bob']
names = tuple(s.title() for s in strings if s)
print(names) # ('John', 'Alice', 'Bob')
```

Notice that the last example creates a *generator object*, which must explicitly be converted to a tuple.

10 Generators

Unlike comprehensions, *generators* use lazy evaluation. Unlike iterators, generators do not require implementing a class implementing the `next()` and `iter()` method (less boilerplate).

Generators are implemented using functions that yield a different result every time they are called:

```
def squares(n):
    for i in range(n):
        yield i ** 2

print(list(squares(6))) # [0, 1, 4, 9, 16, 25]
```

After a value is returned using the `yield` keyword, the generator function stops its execution, but its state is remembered. The execution is continued for the next iteration. This makes it possible for generator functions to hold a state (without using an explicit closure):

```
def factorials(n):
    current = 1
    for i in range(n):
        if i != 0:
            current *= i
        yield current

print(list(factorials(6))) # [1, 1, 2, 6, 24, 120]
```

As seen in the last example of the previous chapter, a generator can be created as a comprehension using parentheses:

```
square_gen = (x ** 2 for x in range(2, 6))
print(next(square_gen)) #  4
print(next(square_gen)) #  9
print(next(square_gen)) # 16
print(next(square_gen)) # 25
print(next(square_gen)) # StopIteration
```

Generators combine the advantages of comprehensions with lazy evaluation. If a sequence is hard to express in terms of `filter` and `map`, and if the task is memory-critical, consider a generator.

11 Partial Application and Currying

Partial application of functions and *currying* are both ways to create new functions based on existing functions. Those techniques are based on closures. (A closure is an inner function returned from a surrounding function, with the inner function having references to the surrounding function.)

11.1 Partial Application

With partial application, only a subset of a function's parameters are set on the first function call. The rest of the parameters are filled in a later call to the partially applied function.

Consider the function `inc_x`, which requires a parameter `x`, and returns a function that increases its parameter by `x`:

```
def inc_x(x):
    def inc(y):
        return x + y
    return inc

numbers = [1, 2, 3]

inc_1 = inc_x(1)
inc_3 = inc_x(3)

print(list(map(inc_1, numbers))) # [2, 3, 4]
print(list(map(inc_3, numbers))) # [4, 5, 6]
```

Partial application is especially helpful if a function has a lot of parameters, like a quadratic function:

$$y = ax^2 + bx + c$$

Such a function is usually defined in terms of the parameters `a`, `b`, and `c`—and applied multiple times using different values for `x`:

```
def quad(a, b, c, x):
    return a*x**2 + b*x + c

def quad_abc(a, b, c):
    def f(x):
        return quad(a, b, c, x)
    return f

xs = range(5)

f = quad_abc(1, 2, 3)
g = quad_abc(2, 0, 1)

print(list(map(f, xs))) # [3, 6, 11, 18, 27]
print(list(map(g, xs))) # [1, 3, 9, 19, 33]
```

The `partial()` functions from the `functools` module provides a more flexible approach that doesn't require defining closures for specific partial applications. The code above can be simplified using `partial()`:

```
from functools import partial

def quad(a, b, c, x):
```

```
    return a*x**2 + b*x + c

xs = range(5)

f = partial(quad, 1, 2, 3)
g = partial(quad, 2, 0, 1)

print(list(map(f, xs))) # [3, 6, 11, 18, 27]
print(list(map(g, xs))) # [1, 3, 9, 19, 33]
```

It is possible to apply a function partially multiple times, until every parameter was filled in:

```
from functools import partial

def quad(a, b, c, x):
    return a*x**2 + b*x + c

xs = range(5)

quad_a = partial(quad, 1)
quad_ab = partial(quad_a, 2)
quad_abc = partial(quad_ab, 3)

print(list(map(quad_abc, xs))) # [3, 6, 11, 18, 27]
```

However, using partial application, the parameters have to be filled in the order as they are defined in the function. It's not possible to just define the `quad` function's parameter `b` and `x`, and leave `a` and `c` undefined.

It is possible though to partially apply a function by setting keyword arguments:

```
from functools import partial

print_csv = partial(print, sep=',')
print_space = partial(print, sep=' ')

names = ['Dilbert', 'Alice', 'Wally']
print_csv(*names)   # Dilbert,Alice,Wally
print_space(*names) # Dilbert Alice Wally
```

11.2 Currying

Python supports currying using third-party libraries such as PyMonad (version 2.4.0):

```
$ pip install --user PyMonad==2.4.0
```

The pymonad module includes the `curry` decorator, which can be used to define functions that can be partially applied without explicit use of `functools.partial`. The number of arguments to be curried needs to be passed to the `curry` decorator:

```
from pymonad.tools import curry

@curry(4)
def quad(a, b, c, x):
    return a*x**2 + b*x + c

xs = range(5)

quad_abc = quad(1, 2, 3)
print(list(map(quad_abc, xs))) # [3, 6, 11, 18, 27]

quad_a = quad(1)
quad_ab = quad_a(2)
quad_abc = quad_ab(3)
print(list(map(quad_abc, xs))) # [3, 6, 11, 18, 27]
```

Notice that curried functions don't come for free; a lot of functions with different argument lists are defined automatically in the background. Calling a curried function is less explicit than using partial application. Make sure the curried nature of a function is made clear to its users by the means of naming, documentation, or convention (writing a module where functions are curried in general).

11.3 Advanced Composition

Functions with a single argument can be composed using a closure:

```
def compose(f, g):
    def fn(x):
        return f(g(x))
    return fn

def increment(x):
    return x + 1

def twice(x):
    return x * 2
```

```
f = compose(twice, increment)

print(f(1)) # 4
print(f(2)) # 6
print(f(3)) # 8
```

Functions with multiple arguments can only be composed as above if partially applied before to turn them into functions accepting a single argument:

```
from functools import partial

def compose(f, g):
    def fn(x):
        return f(g(x))
    return fn

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

def mul(x, y):
    return x * y

increment = partial(add, 1)
twice  = partial(mul, 2)
f = compose(twice, increment)

print(f(1)) # 4
print(f(2)) # 6
print(f(3)) # 8
```

11.3.1 Composing Multiple Functions

Consider the following set of functions `f()`, `g()`, `h()`, and `i()`, which perform the following computations:

- `f(x)`: adds 1 to `x`
- `g(x)`: multiplies `x` by 2
- `h(x)`: computes `x` to the power of 3
- `i(x)`: subtracts 1 from `x`

Thus, `i(h(g(f(x))))` = `(((x + 1) * 2) ^ 3) - 1`. Composing those functions one by one is cumbersome:

```
def compose(f, g):
    def fn(x):
        return f(g(x))
    return fn

def f(x):
    return x + 1

def g(x):
    return x * 2

def h(x):
    return x ** 3

def i(x):
    return x - 1

fn = compose(i, h)
fn = compose(fn, g)
fn = compose(fn, f)

print(fn(1)) # (((1 + 1) * 2) ^ 3) - 1 = 63
```

The composition can be generalized as a reducing operation. The `compose()` function accepts a list of functions to be reduced by composing them pair-wise:

```
from functools import reduce

def compose(*fns):
    def compose_pair(f, g):
        def fn(x):
            return f(g(x))
        return fn
    return reduce(compose_pair, fns)

def f(x):
    return x + 1

def g(x):
    return x * 2

def h(x):
    return x ** 3
```

```
def i(x):
    return x - 1

fn = compose(i, h, g, f)

print(fn(1)) # (((1 + 1) * 2) ^ 3) - 1 = 63
```

Unfortunately, this implementation doesn't work if *no* functions are passed as arguments. The `initializer` argument of `reduce()` can be used to define a default value. A sensible default value, however, depends on the operation to be performed. (For an addition or subtraction, the neutral element is 0, for a multiplicationor a division, the neutral element is 1.) The *identity value* provided by an *identity function* (`lambda x: x`) is the right choice for all cases:

```
from functools import reduce

def compose(*fns):
    def compose_pair(f, g):
        def fn(x):
            return f(g(x))
        return fn
    return reduce(compose_pair, fns, lambda x: x)

fn = compose()
print(fn(37)) # 37
```

12 Functors and Monads

Functors wrap a value and control how functions are applied to that wrapped value. Such wrappers are useful when dealing with values that might be missing, or add new capabilities to existing functions, such as making a function that can only deal with scalar values capable of handling lists of scalar values.

The `oslash` library provides Haskell-style *functors*, *applicatives*, and *monads*:

```
$ pip install oslash==0.6.3
```

- A **functor** wraps a value and controls how function is applied to that wrapped value using the `map()` method or the `%` operator.
- An **applicative** is a special kind of a functor that wraps a function, which can be called by the `apply()` method.
- A **monad** is a special kind of an applicative that also wraps the value returned from a function using its `bind()` method.

Those constructs are crucial in a pure functional programming language like Haskell, where they are needed to deal with errors or side-effects. In Python, those constructs are optional—hence available by third-party libraries such as `oslash`—and can be left away in favour of procedural code.

12.1 Functors

The `Just` functor, which technically is also an applicative and a monad (of which more later), is a wrapper around a value:

```
from oslash import Just

x = Just(3)
print(x) # Just 3
```

Functions cannot be called directly with an instance of `Just` as an argument. Instead, the functor's `map()` method or the `%` operator can be used:

```
from oslash import Just

def twice(x):
    return x * 2

x = Just(3)

y = twice(x) # illegal

y = x.map(twice) # correct
print(y) # Just 6

y = twice % x # correct, but shorter
print(y) # Just 6
```

Notice that the function stands at the left of the `%` operator, and the functor to its right.

The `Nothing` functor does not wrap a value. It is the functional brother of Python's `None` with well-defined behaviour—a function being applied to `Nothing` always returns `Nothing` instead of throwing an exception:

```
from oslash import Nothing

def twice(x):
    return x * 2

x = Nothing()
```

```
print(x) # Nothing

y = twice % x
print(y) # Nothing
```

The `List` functor wraps a list of values and makes it possible that a function that only deals with scalar values can be applied to an entire list—a lot like the higher-order `map()` function (notice that the `twice()` function has to be wrapped by a `Just` functor, of which more in the next section):

```
from oslash import Just, List

def twice(x):
    return x * 2

xs = List.from_iterable([1, 2, 4, 8]) # [1, 2, 4, 8]
print(xs)

f = Just(twice)

ys = f.apply(xs) # [2, 4, 8, 16]
print(ys)
```

12.2 Applicatives

The `Just` functor is, in fact, also an *applicative functor* that wraps a function as a value, or short: an applicative, which provides an `apply()` method:

```
from oslash import Just

def twice(x):
    return x * 2

x = Just(3)
f = Just(twice)

b = f.apply(x)
print(b) # Just 6
```

Notice that both the value `3` and the function `twice()` have been wrapped by a `Just` applicative.

An applicative wrapping a function with more than one parameter returns a partially applied function if the `apply()` method is called on it. The arguments can be filled in one by one:

```
from oslash import Just

def quad(a, b, c, x):
    return (a * x ** 2) + b * x + c

f = Just(quad)
f_a = f.apply(Just(1))
f_ab = f_a.apply(Just(2))
f_abc = f_ab.apply(Just(3))

x = Just(4)

y = f_abc.apply(x)
print(y) # Just 27

y = Just(quad).apply(Just(1)).apply(Just(2)).apply(Just(3)).apply(x)
print(y) # Just 27
```

12.3 Monads

An applicative that also wraps the return value resulting from a call to its wrapped function is called a monad. Its `bind()` method accepts a single parameter—a function returning another monad:

```
from oslash import Just, Nothing

def safe_reciprocal(x):
    if x == 0:
        return Nothing()
    return Just(1/x)

x = Just(4)
y = x.bind(safe_reciprocal)
print(y) # Just 0.25

x = Just(0)
y = x.bind(safe_reciprocal)
print(y) # Nothing
```

13 Useful Libraries

Python's standard library offers a lot of capabilities that support a functional programming style. The `functools` (treated above) and `itertools` (treated in the following section) modules are especially useful for that purpose.

13.1 The `itertools` Module

The `itertools` module provides useful functions to create iterators.

Infinite series of incrementing values can be created using the `count()` function, which requires a `start` value and an optional `step` size:

```
from itertools import count

to_infinite = count(0) # 0, 1, 2, 3, ...
to_infinite = count(0, 10) # 0, 10, 20, 30, ...
```

Infinite or finite repetitions of values can be created using the `repeat()` function, which requires a value `x` to be repeated and an optional limit `n`:

```
from itertools import repeat

infinite_ones = repeat(1) # 1, 1, 1, 1, ...
limited_ones = repeat(1, 3) # 1, 1, 1
```

Series of numbers can be repeated using the `cycle()` function that accepts an iterator to be repeated:

```
from itertools import cycle

one_two_three_ad_nauseam = cycle([1, 2, 3]) # 1, 2, 3, 1, 2, 3
```

Like `zip`, the `zip_longest` function zips together two iterables. Unlike `zip`, it doesn't stop when the shorter iterable is exhausted, but fills in values until the longer iterable is exhausted, too:

```
from itertools import zip_longest

names = ['Dilbert', 'Alice', 'Wally']
ranks = range(1, 6)
ranking = zip_longest(ranks, names, fillvalue='fired')
for rank in ranking:
    print(rank)
```

```
(1, 'Dilbert')
(2, 'Alice')
(3, 'Wally')
(4, 'fired')
(5, 'fired')
```

If a function with `n` parameters is given to the higher-order `map()` function, it expects `n` iterables, too. The higher-order `starmap()` requires a single iterable consisting of `n` tuples instead:

```
from itertools import starmap

inventory = [
    (17, 0.99),
    (32, 0.49),
    (12, 5.49),
    (97, 0.19),
    (13, 2.95),
]

positions = starmap(lambda n, p: n * p, inventory)

for position in positions:
    print(position)
```

```
16.83
15.68
65.88
18.43
38.35
```

The `filterfalse()` higher-order function works like `filter()`, except that it returns the values for which the predicate function returns `False`:

```
from itertools import filterfalse

def is_even(x):
    return x % 2 == 0

numbers = range(10)
even = filter(is_even, numbers)
odd = filterfalse(is_even, numbers)
```

```
print(list(even)) # [0, 2, 4, 6, 8]
print(list(odd))  # [1, 3, 5, 7, 9]
```

The `accumulate()` function works like `sum()`, but keeps a running total:

```
from itertools import accumulate

xs = range(5)
sums = accumulate(xs)

print(list(xs))   # [0, 1, 2, 3, 4]
print(list(sums)) # [0, 1, 3, 6, 10]
```

Two or more iterables can be joined together using the `chain()` function:

```
from itertools import chain

xs = range(3)
ys = range(3, 6)
zs = range(6, 9)

print(list(chain(xs, ys, zs))) # [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

An iterable can be turned into `n` iterables with the same underlying values using the `tee()` function:

```
from itertools import tee

xs = range(5)

a, b, c = tee(xs, 3)
print(list(a)) # [0, 1, 2, 3, 4]
print(list(b)) # [0, 1, 2, 3, 4]
print(list(c)) # [0, 1, 2, 3, 4]
```

The `takewhile()` function works like `filter()`, but stops after the first item fails the predicate function. The `dropwhile()` function ignores values until the first item matches the predicate function:

```
from itertools import takewhile, dropwhile

def is_even(x):
    return x % 2 == 0

numbers = [0, 2, 4, 6, 7, 8, 10, 11]
```

```
left = takewhile(is_even, numbers)
right = dropwhile(is_even, numbers)

print(list(left))  # [0, 2, 4, 6]
print(list(right)) # [7, 8, 10, 11]
```

Notice that the value `11` is included in the `right` list, even though it wouldn't match the `is_even()` predicate function.

See the `itertools` and `functools` documentation pages for more details and additional useful functions.

13.2 Third-Party Libraries

The following third-party libraries have been introduced in this text:

- PyMonad providing functional programming techniques the Python standard library doesn't.
- OSlash providing functors, applicatives, and monads.