Chapter 3

Functional Programming

In general, the most used programming approach for introductory courses is the procedural one, where we organize the code as a list of instructions that tell the computer how to process a given input. In chapter 1 we introduced the Object-Oriented paradigm, where programs represent the functionalities by objects-interaction through their attributes and methods that modify the attributes or states of each object. In the functional approach, we organize our code as a set of related functions that we can pass as arguments, modify or return them. Functions' outputs can be inputs to other functions. Functions' scope is only the code contained inside them; they do not use or modify any data outside their scope

The functional programming forces us to write a modular solution, breaking into apart our tasks into small pieces. It is advantageous during debugging and testing because the wrote functions are small and easy to read. In debugging, we can quickly isolate errors in a particular function. When we test a program, we see each function as a unit, so we only need to create the right input and then check its corresponding output.

Python is a multi-paradigm programming language, *i. e*, our solutions could be written simultaneously either in a procedural way, using object-oriented programming or applying a functional approach. In this chapter, we explain the core concepts of functional programming in Python and how we develop our applications using this technique.

3.1 Python Functions

There are many functions already implemented in Python, mainly to simplify and to abstract from calculations that we can apply to several different types of classes (duck typing). We recommend the reader to check the complete list of built-in functions in [1]. Let's see a few examples:

Len

Returns the number of elements in any container (list, array, set, etc.)

```
print(len([3, 4, 1, 5, 5, 2]))
print(len({'name': 'John', 'lastname': 'Smith'}))
print(len((4, 6, 2, 5, 6)))
```

```
6
2
5
```

This function comes implemented as the internal method (`__len__`) in most of Python default classes:

```
print([3, 4, 1, 5, 5, 2].__len__())
print({'name': 'John', 'lastname': 'Smith'}.__len__())
```

```
6
2
```

When `len(MyObject)` is called, it actually calls the method `MyObject.__len__()`:

```
print(id([3, 4, 1, 5, 5, 2].__len__()))
print(id(len([3, 4, 1, 5, 5, 2])))
```

```
4490937616
4490937616
```

We can also override the `__len__` method. Suppose we want to implement a special type of list (`MyList`) such that `len(MyList())` returns the length of the list ignoring repeated occurences:

```
from collections import defaultdict


class MyList(list):
    def __len__(self):
        # Each time this method is called with a non-existing key, the
        # key-value pair is generated with a default value of 0
        d = defaultdict(int)
        # This value comes from calling "int" without arguments. (Try
```

```
10          # typing int() on Python's console)
11
12          # Here we call the original method from the super-class list
13          for i in range(list.__len__(self)):
14              d.update({self[i]: d[self[i]] + 1})
15
16          # Here we call d's (a defaultdict) len method
17          return len(d)
18
19
20  L = MyList([1, 2, 3, 4, 5, 6, 6, 7, 7, 7, 7, 2, 2, 3, 3, 1, 1])
21  print(len(L))
```

```
7
```

```
1   from collections import defaultdict
2
3   # Another way of achieving the same behaviour
4   class MyList2(list):
5       def __len__(self):
6           d = defaultdict(int)
7
8           for i in self:  # Here we iterate over the items contained in the object
9               d.update({i: d[i] + 1})
10
11          return len(d)
12
13
14  L = MyList2([1, 2, 3, 4, 5, 6, 6, 7, 7, 7, 7, 2, 2, 3, 3, 1, 1])
15  print(len(L))
```

```
7
```

```
1   # Yet another way
2   class MyList3(list):
3       def __len__(self):
4           d = set(self)
```

```
5           return len(d)
6
7   L = MyList3([1, 2, 3, 4, 5, 6, 6, 7, 7, 7, 7, 2, 2, 3, 3, 1, 1])
8   print(len(L))
```

```
7
```

Getitem

Declaring this function within a class allows each instance to become iterable (you can iterate over the object, soon we delve further into iterable objects). Besides allowing iteration, the `__getitem__` method lets us use indexation over the objects:

```
1   class MyClass:
2       def __init__(self, word=None):
3           self.word = word
4
5       def __getitem__(self, i):
6           return self.word[i]
7
8
9   p = MyClass("Hello World")
10  print(p[0])
11
12  [print(c) for c in p]
13
14  (a, b, c, d) = p[0:4]
15  print(a, b, c, d)
16  print(list(p))
17  print(tuple(p))
```

```
H
H
e
l
l
o
```

```
W
o
r
l
d
H e l l
['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']
('H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd')
```

Reversed

The `reversed` function takes a sequence as input and returns a copy of the sequence in reversed order. We can also customize the function by overriding the `__reversed__` method in each class. If we do not customize this function, the built-in will be used, by iterating once from `__len__` to `0` using the `__getitem__` method.

```
1  a_list = [1, 2, 3, 4, 5, 6]
2
3
4  class MySequence:
5      # given that we are not overriding the __reversed__ method, the built-in
6      # will be used (iterating with __getitem__ and __len___)
7      def __len__(self):
8          return 9
9
10     def __getitem__(self, index):
11         return "Item_{0}".format(index)
12
13
14 class MyReversed(MySequence):
15     def __reversed__(self):
16         return "Reversing!!"
17
18
19 for seq in a_list, MySequence(), MyReversed():
20     print("\n{} : ".format(seq.__class__.__name__), end="")
```

```
21      for item in reversed(seq):
22          print(item, end=", ")
```

```
list : 6, 5, 4, 3, 2, 1,
MySequence : Item_8, Item_7, Item_6, Item_5, Item_4, Item_3, Item_2, Item_1,
Item_0,
MyReversed : R, e, v, e, r, s, i, n, g, !, !,
```

Enumerate

The `enumerate` method creates an iterable of tuples, where the first item in each tuple is the index and the second is the original object in the corresponding index.

```
1 a_list = ["a", "b", "c", "d"]
2
3 for i, j in enumerate(a_list):
4     print("{}: {}".format(i, j))
5
6 print([pair for pair in enumerate(a_list)])
7
8 # We create a dictionary using the index given by "enumerate" as key
9 print({i: j for i, j in enumerate(a_list)})
```

```
0: a
1: b
2: c
3: d
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
{0: 'a', 1: 'b', 2: 'c', 3: 'd'}
```

Zip

This function takes n sequences (two or more) and generates a sequence of n-tuples with the relative objects in each sequence:

```
1 variables = ['name', 'lastname', 'email']
2 p1 = ["John", 'Smith', 'js1@hotmail.com']
```

```
3 p2 = ["Thomas", 'White', 'thwh@gmail.com']
4 p3 = ["Jeff", 'West', 'jwest@yahoo.com']
5
6 contacts = []
7 for p in p1,p2,p3:
8     contact = zip(variables, p)
9     contacts.append(dict(contact))
10
11 for c in contacts:
12     print("Name: {name} {lastname}, email: {email}".format(**c))
13     #**c passes the dictionary as a keyworded list of arguments
```

```
Name: John Smith, email: js1@hotmail.com
Name: Thomas White, email: thwh@gmail.com
Name: Jeff West, email: jwest@yahoo.com
```

The `zip` function is also its own inverse:

```
1 A = [1, 2, 3, 4]
2 B = ['a', 'b', 'c', 'd']
3
4 zipped = zip(A, B)
5 zipped = list(zipped)
6 print(zipped)
7 unzipped = zip(*zipped)
8 unzipped = list(unzipped)
9 print(unzipped)
```

```
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
[(1, 2, 3, 4), ('a', 'b', 'c', 'd')]
```

Comprehensions

Defining a set of elements by comprehension allows you to explicitly describe the content without enumerating each one of the elements. In Python we can do this as follows:

List comprehensions:

```
a_list = ['1', '4', '55', '65', '4', '15', '90']
int_list = [int(c) for c in a_list]
print("int_list:", int_list)

int_list_2d = [int(c) for c in a_list if len(c) > 1]
print("int_list_2d:", int_list_2d)
```

```
int_list: [1, 4, 55, 65, 4, 15, 90]
int_list_2d: [55, 65, 15, 90]
```

Sets and Dictionary comprehensions:

```
from collections import namedtuple

#namedtuple is a tuple subclass that has fields (with arbitrary names),
#which can be accessed as tuple.field
Movie = namedtuple("Movie", ["title", "director", "genre"])
movies = [Movie("Into the Woods", "Rob Marshall", "Adventure"),
          Movie("American Sniper", "Clint Eastwood", "Action"),
          Movie("Birdman", "Alejandro Gonzalez Inarritu", "Comedy"),
          Movie("Boyhood", "Richard Linklater", "Drama"),
          Movie("Taken 3", "Olivier Megaton", "Action"),
          Movie("The Imitation Game", "Morten Tyldum", "Biography"),
          Movie("Gone Girl", "David Fincher", "Drama")]

# set comprehension
action_directors = {b.director for b in movies if b.genre == 'Action'}
print(action_directors)
```

```
{'Clint Eastwood', 'Olivier Megaton'}
```

We can create dictionaries from search results:

```
action_directors_dict = {b.director: b for b in movies if b.genre == 'Action'}
print(action_directors_dict)
print(action_directors_dict['Olivier Megaton'])
```

```
{'Clint Eastwood': Movie(title='American Sniper', director='Clint Eastwood',
```

```
genre='Action'), 'Olivier Megaton': Movie(title='Taken 3',
director='Olivier Megaton', genre='Action')}
Movie(title='Taken 3', director='Olivier Megaton', genre='Action')
```

Iterables and Iterators

A *iterable* is any object over which you can iterate. Therefore, we can use any iterable on the right side of a *for* loop. We can iterate an infinite amount of times over an iterable, just like with lists. This type of objects must contain the `__iter__` method.

A *iterator* is an object that iterates over an iterable. These objects contain the `__next__` method, which will return the next element each time we call it. The object returned by the `__iter__` method must be an iterator. Let's see the following example:

```
1 x = [11, 32, 43]
2 for c in x:
3     print(c)
4 print(x.__iter__)
5 next(x)  # Lists are not iterators
```

```
11
32
43
<method-wrapper '__iter__' of list object at 0x10bef2e48>
'list' object is not an iterator
```

As we can see above, list objects are not iterators, but we can get an iterator over a list by calling the `iter` method.

```
1 y = iter(x)  # equivalent to x.__iter__
2 print(next(y))
3 print(next(y))
4 print(next(y))
```

```
11
32
43
```

```
1 class Card:
```

```
2      FACE_CARDS = {11: 'J', 12: 'Q', 13: 'K'}
3
4      def __init__(self, value, suit):
5          self.suit = suit
6          self.value = value if value <= 10 else Card.FACE_CARDS[value]
7
8      def __str__(self):
9          return "%s %s" % (self.value, self.suit)
10
11
12 class Deck:
13     def __init__(self):
14         self.cards = []
15         for s in ['Spades', 'Diamonds', 'Hearts', 'Clubs']:
16             for v in range(1, 14):
17                 self.cards.append(Card(v, s))
18
19
20 for c in Deck().cards:
21     print(c)
```

```
1 Spades
  ...
K Spades
1 Diamonds
  ...
K Diamonds
1 Hearts
  ...
K Hearts
1 Clubs
  ...
K Clubs
```

Even though a `Deck` instance contains many cards, we can not iterate directly over it, only over `Deck().cards` (which corresponds to a list, an iterable object). Suppose we want to iterate over `Deck()` directly. In order to do so,

we should define the `__iter__` method.

```
class Deck:
    def __init__(self):
        self.cards = []
        for p in ['Spades', 'Diamonds', 'Hearts', 'Clubs']:
            for n in range(1, 14):
                self.cartas.append(Card(n, p))

    def __iter__(self):
        return iter(self.cards)


for c in Deck():
    print(c)
```

```
1 Spades
  ...
K Spades
1 Diamonds
  ...
K Diamonds
1 Hearts
  ...
K Hearts
1 Clubs
  ...
K Clubs
```

Let's see an example of how to create an iterator:

```
class Fib:
    def __init__(self):
        self.prev = 0
        self.curr = 1

    def __iter__(self):
        return self
```

```
8
9       def __next__(self):
10          value = self.curr
11          self.curr += self.prev
12          self.prev = value
13          return value
14
15
16  f = Fib()
17  N = 10
18  l = [next(f) for i in range(N)]
19  print(l)
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Python's module "`itertools`" provides many iterators. Here are some examples:

```
1   import itertools
2
3   letters = ['a', 'b', 'c', 'd', 'e', 'f']
4   bools = [1, 0, 1, 0, 0, 1]
5   nums = [23, 20, 44, 32, 7, 12]
6   decimals = [0.1, 0.7, 0.4, 0.4, 0.5]
7
8   # Iterates indefinitely over letters.
9   colors = itertools.cycle(letters)
10  print(next(colors))
11  print(next(colors))
12  print(next(colors))
13  print(next(colors))
14  print(next(colors))
15  print(next(colors))
16  print(next(colors))
17  print(next(colors))
18  print(next(colors))
19  print(next(colors))
```

```
a
b
c
d
e
f
a
b
c
d
```

```
# Iterates across all the iterables in the arguments consecutively.
for i in itertools.chain(letters, bools, decimals):
    print(i, end=" ")
```

```
a b c d e f 1 0 1 0 0 1 0.1 0.7 0.4 0.4 0.5
```

```
# Iterates over the elements in letters according to the condition in bools.
for i in itertools.compress(letters, bools):
    print(i, end=" ")
```

```
a c f
```

Generators

Generators are a particular type of iterators; they allow us to iterate over sequences without the need to save them in a data structure, avoiding unnecessary memory usage. Once we finish the iteration over a generator, the generator disappears. It is useful when you want to perform calculations on sequences of numbers that only serve a purpose in a particular calculation. The syntax for creating generators is very similar to a list comprehension, but instead of using square brackets `[]`, we use parentheses `()`:

```
from sys import getsizeof

# using parenthesis indicates that we are creating a generator
a = (b for b in range(10))

print(getsizeof(a))
```

```
7
8   c = [b for b in range(10)]
9
10  # c uses more memory than a
11  print(getsizeof(c))
12
13  for b in a:
14      print(b)
15
16  print(sum(a)) # the sequence has disappeared
```

```
72
192
0
1
2
3
4
5
6
7
8
9
0
```

Example: Suppose that the archive `logs.txt` contains the following lines::

```
Abr 13, 2014 09:22:34
Jun 14, 2014 08:32:11
May 20, 2014 10:12:54
Dic 21, 2014 11:11:62
WARNING We are about to have a problem.
WARNING Second Warning!
WARNING This is a bug
WARNING Be careful
```

```
inname, outname = "logs.txt", "logs_out.txt"

with open(inname) as infile:
    with open(outname, "w") as outfile:
        warnings = (l.replace('WARNING', '') for l in infile if 'WARNING' in l)
        for l in warnings:
            outfile.write(l)
```

The contents of `logs_out.txt` should read as goes:

```
We are about to have a problem.
Second Warning!
This is a bug
Be careful
```

Generator Functions

Python functions are also able to work as generators, through the use of the `yield` statement. `yield` replaces `return`, which besides being responsible for returning a value, it assures that the next function call will be executed starting from that point. In other words, we work with a method that once it “returns” a value through `yield`, it transfers the control back to the outer scope only temporarily, waiting for a successive call to “generate” more values. Calling a generator function creates a generator object. However, this does not start the execution of the function:

```
def dec_count(n):
    print("Counting down from {}".format(n))
    while n > 0:
        yield n
        n -= 1
```

The function is only executed once we call the generated object’s `__next__` method:

```
x = dec_count(10)   # Note that this does not print anything
print("{}\n".format(x))   # here we are printing the object itself
y = dec_count(5)
print(next(y))
print(next(y))
print(next(y))
```

```
7 print(next(y))
```

```
<generator object dec_count at 0x1080464c8>

Counting down from 5
5
4
3
2
```

```
1  def fibonacci():
2      a, b = 0, 1
3      while True:
4          yield b
5          a, b = b, a + b
6
7
8  f = fibonacci()
9  print(next(f))
10 print(next(f))
11 print(next(f))
12 print(next(f))
13 print(next(f))
14 print(next(f))
15 g1 = [next(f) for i in range(10)]
16 print(g1)
17 g2 = (next(f) for i in range(10))
18 for a in g2:
19     print(a)
```

```
1
1
2
3
5
8
[13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

```
1597
2584
4181
6765
10946
17711
28657
46368
75025
121393
```

```
import numpy as np

def maximum(values):
    temp_max = -np.infty
    for v in values:
        if v > temp_max:
            temp_max = v
        yield temp_max

elements = [10, 14, 7, 9, 12, 19, 33]
res = maximum(elements)
print(next(res))
print(next(res))
print(next(res))
print(next(res))
print(next(res))
print(next(res))
print(next(res))
print(next(res))  # we've run out of list elements!
```

```
10
14
14
14
14
```

```
19
33
```

We can also interact with a function by sending messages. The `send` method allows us to send a value to the generator. We can assign that value to a variable by using the `yield` statement. When we write `a = yield`, we are assigning to the variable `a` the value sent by the `send` method. When we write `a = yield b`[1], besides assigning the sent value to `a`, the function is returning the object `b`:

```
1 def mov_avg():
2     print("Entering ...")
3     total = float((yield))
4     cont = 1
5     print("total = {}".format(total))
6     while True:
7         print("While loop ...")
8         # Here i receive the message and also the yield returns total/count
9         i = yield total / cont
10        cont += 1
11        total += i
12        print("i = {}".format(i))
13        print("total = {}".format(total))
14        print("cont = {}".format(cont))
```

Note that the code must run until the first `yield` in order to start accepting values through `send()`. Hence it is always necessary to call `next()` (or `send(None)`) after having created the generator to be able to start sending data:

```
1 m = mov_avg()
2 print("Entering to the first next")
3 next(m)  # We move to the first yield
4 print("Leaving the first next")
5 m.send(10)
6 print("Entering to send(5)")
7 m.send(5)
8 print("Entering to send(0)")
9 m.send(0)
```

[1]Use of parentheses around `yield b` may be needed if you want to operate over the sent value

```
10 print("Entering to second send(0)")
11 m.send(0)
12 print("Entering to send(20)")
13 m.send(20)
```

```
Entering to the first next
Entering ...
Leaving the first next
total = 10.0
While loop ...
Entering to send(5)
i = 5
total = 15.0
cont = 2
While loop ...
Entering to send(0)
i = 0
total = 15.0
cont = 3
While loop ...
Entering to second send(0)
i = 0
total = 15.0
cont = 4
While loop ...
Entering to send(20)
i = 20
total = 35.0
cont = 5
While loop ...
```

The following example shows how to perform the UNIX grep command by using a generator function.

```
1 def grep(pattern):
2     print("Searching for %s" % pattern)
3     while True:
4         line = yield
```

```
5           if pattern in line:
6               print(line)
7
8
9   o = grep("Hello")  # creating the object won't execute the function yet
10  next(o)  # Move on to the first yield, "Searching for ..." will be printed
11
12  o.send("This line contains Hello")
13  o.send("This line won't be printed")
14  o.send("This line will (because it contains Hello :) )")
15  o.send("This line won't be shown either")
```

```
Searching for Hello
This line contains Hello
This line will (because it contains Hello :) )
```

Lambda Functions

Lambda functions are short methods created “on the fly”. Their expressions are always returned (no need for the “`return`” statement). Examples:

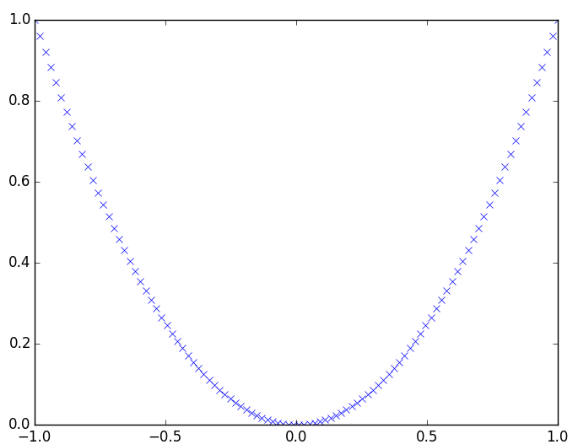
```
1   strings = ["ZZ", "YY", "bb", "aa"]
2   print("Simple sort:", sorted(strings))
3
4   # If we want to sort according to the lowercase values:
5   def lower(s):
6       return s.lower()
7
8   print("Lower sort: ", sorted(strings, key=lower))
9
10  # The same result can be achieved with a lambda function:
11  print("Lambda sort:", sorted(strings, key=lambda s: s.lower()))
```

```
Simple sort: ['YY', 'ZZ', 'aa', 'bb']
Lower sort:  ['aa', 'bb', 'YY', 'ZZ']
Lambda sort: ['aa', 'bb', 'YY', 'ZZ']
```

Map

The `map` function takes a function and an iterable and returns a generator that results from applying the function to each value on the iterable. `map(f, iterable)` is equivalent to `[f(x) for x in iterable]`

```
1  from matplotlib import pyplot as plt
2  import numpy as np
3
4  pow2 = lambda x: x ** 2
5  # Creates a 100 element numpy array, ranging evenly from -1 to 1
6  t = np.linspace(-1., 1., 100)
7  plt.plot(t, list(map(pow2, t)), 'xb')
8  plt.show()
```



We can also apply `map` to more than one iterable at once:

```
1  a = [1, 2, 3, 4]
2  b = [17, 12, 11, 10]
3  c = [-1, -4, 5, 9]
4
5  c1 = list(map(lambda x, y: x + y, a, b))
6
7  c2 = list(map(lambda x, y, z: x + y + z, a, b, c))
8
9  c3 = list(map(lambda x, y, z: 2.5 * x + 2 * y - z, a, b, c))
10
```

```
11 print(c1)
12 print(c2)
13 print(c3)
```

```
[18, 14, 14, 14]
[17, 10, 19, 23]
[37.5, 33.0, 24.5, 21.0]
```

Filter

`filter(f, sequence)` returns a new sequence that includes all the values from the original sequence in which the result of applying `f(value)` was `True`. Function `f` should always return a boolean value:

```
1 f = fibonacci() # Defined before
2 fib = [next(f) for i in range(11)]
3 odds = list(filter(lambda x: x % 2 != 0, fib))
4 print("Odd:", odds)
5
6 even = list(filter(lambda x: x % 2 == 0, fib))
7 print("Even:", even)
```

```
Odd: [1, 1, 3, 5, 13, 21, 55, 89]
Even: [2, 8, 34]
```

Reduce

`reduce(f, [s1,s2,s3,...,sn])` returns the result of applying `f` over the sequence `[s1,s2,s3,...,sn]` as follows: `f(f(f(f(s1,s2),s3),s4),s5),...` The following code shows an example:

```
1 from functools import reduce
2 import datetime
3
4 reduce(lambda x, y: x + y, range(1, 10))
5
6 # Lets compute the length of a file's longest line
7 # rstrip returns a string copy that has no trailing spaces
8 #(or the character specified)
```

```
9  # eg: "Hello...".rstrip(".") returns "Hello"
10
11 t = datetime.datetime.now()
12 r1 = reduce(max, map(lambda l: len(l.rstrip()), [line for line
13                                                   in open('logs_out.txt')]))
14 print("reduce time: {}".format(datetime.datetime.now() - t))
15 print(r1)
16
17 # Another way of doing the same with generator comprehensions and numpy
18 t = datetime.datetime.now()
19 r2 = max((len(line.rstrip()) for line in open('logs_out.txt')))
20 print("max(generator) time: {}".format(datetime.datetime.now() - t))
21 print(r2)
22
23 # To visualize the lines of the file
24 for line in open('logs_out.txt'):
25     print(line.rstrip())
```

```
reduce time: 0:00:00.000224
31
max(generator) time: 0:00:00.000158
31
We are about to have a problem.
Second Warning!
This is a bug
Be careful
```

3.2 Decorators

Decorators allow us to take an already implemented feature, add some behavior or additional data and return a new function. We can see decorators as functions that receive any function `f1` and return a new function `f2` with a modified behaviour. If our decorator is called `dec_1`, in order to modify a function and assign it to the same name, we should simply write `f1 = dec_1(f1)`.

Our function `f1` now contains the new data and aggregate behavior. One benefit of decorators is that we avoid the need to modify the code of the original function (and if we want the original version of the function, we simply remove the call to the decorator). It also avoids creating a different function with a different name (this would imply modifying all the calls to the function you want to change).

Take the following inefficient recursive implementation of a function that returns the Fibonacci numbers:

```
import datetime


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


n = 35
t1 = datetime.datetime.now()
print(fib(n))
print("Execution time: {}".format(datetime.datetime.now() - t1))
```

```
9227465
Execution time: 0:00:06.462758
```

A more efficient implementation might try to "store" numbers already calculated in the Fibonacci sequence. We can use a decorator that receives the `fib` function, adds memory to it and checks for the existence of that number in a previous call:

```
def efficient_fib(f):  # recieves a function as an argument
    data = {}

    def func(x):  # this is the new function to be returned
        if x not in data:
            data[x] = f(x)  # the function recieved as an argument
```

```
7                               #is now called
8           return data[x]
9
10      return func
11
12  # we use the decorator.
13  fib = efficient_fib(fib)
14  # The fib function is now "decorated" by the function
15  #"efficient_fib"
16  t1 = datetime.datetime.now()
17
18  # We still use the same function name, there is no need
19  #to call the new function
20  print(fib(n))
21  print("Execution time: {}".format(datetime.datetime.now() - t1))
```

```
9227465
Execution time: 0:00:00.000144
```

Using Python's alternative notation for decorators:

```
1  @eficient_fib
2  def fib(n):
3      if n == 0:
4          return 0
5      elif n == 1:
6          return 1
7      else:
8          return fib(n-1) + fib(n-2)
9
10 n = 35
11 t1 = datetime.datetime.now()
12 print(fib(n))
13 print("Execution time: {}".format(datetime.datetime.now()-t1))
```

```
9227465
Execution time: 0:00:00.000038
```

We can use a hierarchy of decorators and receive parameters for decoration. A generic way to do this is:

```
def mydecorator(function):
    def _mydecorator(*args, **kw):
        # Do stuff here before calling the original function
        # call the function
        res = function(*args, **kw)
        # Do more stuff after calling the function
        return res

    # return the sub-function
    return _mydecorator
```

```
import time
import hashlib
import pickle

cache = {}


def is_obsolete(entry, duration):
    return time.time() - entry['time'] > duration


def compute_key(function, args, kw):

    key = pickle.dumps((function.__name__, args, kw))
    # returns the pickle representation of an object as a byte object
    # instead of writing it on a file

    # creates a key from the "frozen" key generated in the last step
    return hashlib.sha1(key).hexdigest()


def memoize(duration=10):
    def _memoize(function):
        def __memoize(*args, **kw):
```

```
25              key = compute_key(function, args, kw)
26
27              # do we have the value on cache?
28              if key in cache and not is_obsolete(cache[key], duration):
29                  print('we already have the value')
30                  return cache[key]['value']
31
32              # if we didn't
33              print('calculating...')
34              result = function(*args, **kw)
35              # storing the result
36              cache[key] = {'value': result, 'time': time.time()}
37              return result
38
39          return __memoize
40
41      return _memoize
```

```
1  @memoize(0.0001)
2  def complex_process(a, b):
3      return a + b
4
5  # This is the same as calling
6  # complex_process = memoize(0.0001)(complex_process)
7  # after defining the function
8
9  print(complex_process(2, 2))
10 print(complex_process(2, 1))
11 print(complex_process(2, 2))
12 print(complex_process(2, 2))
13 print(complex_process(2, 2))
14 print(complex_process(2, 2))
15 print(complex_process(2, 2))
16 print(complex_process(2, 2))
17 print(complex_process(2, 2))
```

```
calculating...
4
calculating...
3
we already have the value
4
we already have the value
4
we already have the value
4
calculating...
4
we already have the value
4
we already have the value
4
we already have the value
4
```

Here an example of an access protection decorator:

```
class User:
    def __init__(self, roles):
        self.roles = roles


class Unauthorized(Exception):
    pass


def protect(role):
    def _protect(function):
        def __protect(*args, **kw):
            user = globals().get('user')
            if user is None or role not in user.roles:
                raise Unauthorized("Not telling you!!")  # exceptions coming soon!
```

```
16                return function(*args, **kw)
17
18            return __protect
19
20        return _protect
21
22
23    john = User(('admin', 'user'))
24    peter = User(('user',))
25
26
27    class Secret:
28        @protect('admin')
29        def pisco_sour_recipe(self):
30            print('Use lots of pisco!')
31
32
33    s = Secret()
34    user = john
35    s.pisco_sour_recipe()
36    user = peter
37    s.pisco_sour_recipe()
```

```
Use lots of pisco!
__main__.Unauthorized: Not telling you!!
```

We can also decorate classes in the same way we decorate functions:

```
1   # Lets suppose we want to decorate a class such that it prints a warning
2   # when we try to spend more than what we've got
3   def add_warning(cls):
4       prev_spend = getattr(cls, 'spend')
5
6       def new_spend(self, money):
7           if money > self.money:
8               print("You are spending more than what you have, "
9                     "a debt has been generated in your account!!")
```

```
10          prev_spend(self, money)
11
12      setattr(cls, 'spend', new_spend)
13      return cls
14
15
16  @add_warning
17  class Buy:
18      def __init__(self, money):
19          self.money = money
20          self.debt = 0
21
22      def spend(self, money):
23          self.money -= money
24          if self.money < 0:
25              self.debt = abs(self.money)  # the debt is considered positive
26              self.money = 0
27          print("Current Balance = {}".format(self.money))
28
29
30  b = Buy(1000)
31  b.spend(1200)
```

```
You are spending more than what you have, a debt has been generated in your account!!
Current Balance = 0
```

3.3 Hands-On Activities

Activity 3.1

In this activity, we have a file called *Report.txt* contains information about patients that attended to the city hospital during one year. Each line refers to the fields *year of attention*, *month of attention*, *day of the week*, *assigned color*, *time of attention* and *release reason*, *separated by tabs*. The *assigned color* shows how critical is the medical condition of the patient. The color code is from most to less critical: *blue*, *red*, *orange*, *yellow* and *green*. Your task is to create an application able to read the information from the file and generate the necessary classes and objects using this information. To read the file, you have to create a *generator function* that *yields* each line of the file one by one. You

will also have to create a class called `Report`. An instance of this class has to keep the list of all the patients. The instance of `Report` has to be a *iterable*, such that, iterations over it must return each patient in its list of patients. Also, the `Report` class have to contain a function that given a color it returns all the patients assigned to this color. The returned list must be created using comprehension.

Each patient instance have all the information contained in the file *Report.txt* plus a personal `id` generated by using a *generator* function. Also, each patient must be able to be printed showing all his personal information, including the *id* assigned by the system. After reading all the file, your application must print out all the patients.

Activity 3.2

A soccer team needs to hire some new players to improve their results in the next sports season. They bought a data file called `players.txt` that contains information with most of the soccer players in the league. The team asks your help to process the data file and get valuable information for hiring. Each line of the archive contains information about one player in comma separated values format (CSV):

```
names; last_name_1; last_name_2; country; footedness; birth_day; birth_month;
                birth_year; number_of_goals; high_cm, weight_kg
```

Using mostly only **map**, **reduce** and **filter**, you must perform the following tasks:

1. Read the data file `players.txt` using `map`. Generate a list of tuples, where each tuple contains the data from each player.
2. For each of the queries below, create a function `name_query(list_tuples)` that returns the following:
   a) **Has the name:** Returns a list of tuples with the information about the players that have a defined name or last name (1 or 2)
   b) **Lefty-Chileans:** Returns a list of tuples with the information about all the lefty players from Chile.
   c) **Ages:** Returns a list of tuples with the format `(names, last_name_1, age)` of every player. For simplicity, just use the year of birth to calculate the age.
   d) **Sub-17:** Returns a list of tuples with the format `(names, last_name_1)` if every players that have 17 years old or less (hint: use the previous result).
   e) **Top Scorer:** Returns a tuple with all the information from the top scorer player. You can assume that exists just one top scorer.

f) **Highest obesity risk:** Returns a tuple with the format `(names, last_name_1, high_cm, weight_kg, bmi)`. Where `bmi` is the body mass index, calculated as the body mass divided by the square of the body height (kg/m^2)

Using all the coded functions, print the results of all the queries.

Activity 3.3

The owner of a hamburger store wants you to implement an upgrade for the current production management system. They have the main class that models each product; you have to create a decorator for that class such that you save every newly created instance in a list (belonging to that class) called `instances`.

The upgrade has to allow the system to compare the produced hamburgers according to a customizable attribute that may vary in the future. Your goal is to include a decorator called `compare_by` that receives as a parameter the name of the attribute used for the comparison, such that it allows for comparing instances. We have to make all the possible comparisons though the operators: $<$, $>$, $=$, $\leq$, $\geq$. For example, `hamburger1` $>$ `hamburger2` returns `True` if the attribute specified in the decorator for hamburger1 is higher than the same attribute in hamburger2.

The current function used by the management system to calculate the final price uses a fix tax value. However, The Government will change the tax percentage in the future from 19% to 23%. Your upgrade should also change the way the system calculates the amount of the tax applied to each sale. Unfortunately, the function cannot be directly modified. Therefore, one of the most suitable solutions is to change the behavior of the function using a decorator, called `change_tax`.

Tips and examples

- **Retrieving and modifying attributes**

  You can use `getattr` and `setattr` to retrieve and to update the attributes of an object.

```
1 # Access
2 old_value = getattr(object, 'attribute_name')
3 # Modify
4 setattr(object, 'attribute_name', new_value)
```

- **Decorating a class**

  The following example shows a decorator that modifies a class method such that it prints out a message every time we call it.

```python
def call_alert(method_name):
    def _decorator(cls):
        method = getattr(cls, method_name)

        def new_method(*args, **kwargs):
            print('Calling the method!')
            return method(*args, **kwargs)

        setattr(cls, method_name, new_method)
        return cls

    return _decorator

#Here we apply it to a test class:
@call_alert('walk')
class Test:
    def walk(self):
        return 'I am walking'

if __name__ == "__main__":
    t = Test()
    print(t.walk())
```

The following script shows the current production management system used by the store. Add your decorators at the beginning and then, decorate the class `Hamburger` and the function `price_after_tax`:

```python
class Hamburger:

    def __init__(self, high, diameter, meat_quantity):
        self.high = high
        self.diameter = diameter
        self.meat_quantity = meat_quantity

    def __repr__(self):
        return ('Hamburger {0} cms high, '
                '{1} cm of diameter and '
```

```
11                    '{2} meat quantity').format(self.high, self.diameter,
12                                               self.meat_quantity)
13
14  def price_after_tax(price_before_tax):
15      return (price_before_tax * 1.19 + 100)
16
17
18  if __name__ == "__main__":
19      hamburger1 = Hamburger(10, 15, 2)
20      hamburger2 = Hamburger(7, 10, 3)
21      hamburger3 = Hamburger(10, 9, 2)
22
23      print(hamburger2 > hamburger1)
24      print(hamburger2 == hamburger3)
25      print(hamburger1 < hamburger3)
26
27      print(Hamburger.instances)
28      hamburger4 = Hamburger(12, 20, 4)
29      print(Hamburger.instances)
30      print(price_after_tax(2000))
```