UNIVERSITÄT BIELEFELD
Faculty of Technology

Programming

Advanced Programming

Harsha Manjunath

Faculty of Technology,

Bielefeld University

Functional Programming

Lazy Evaluation

Object-oriented Programming

Functional Programming

Recursion

A resursive function is a function that calls itself. This example is from Lecture 03.

In [1]:
```
def fun(x):
    if len(x) > 1:
        return fun(x[1:])
    return x

fun([1, 2, 3, 4])
```

Out[1]:
```
[4]
```

Functions as objects

In Python everything is an object, including functions:

In [2]:
```
def makeList(a, b):
    return [a, b]

myVariable = makeList
```

We can now call the variable that points to the function:

In [3]:
```
myVariable(1, 2)
```

Out[3]:
```
[1, 2]
```

This also allows us to pass functions on to other functions:

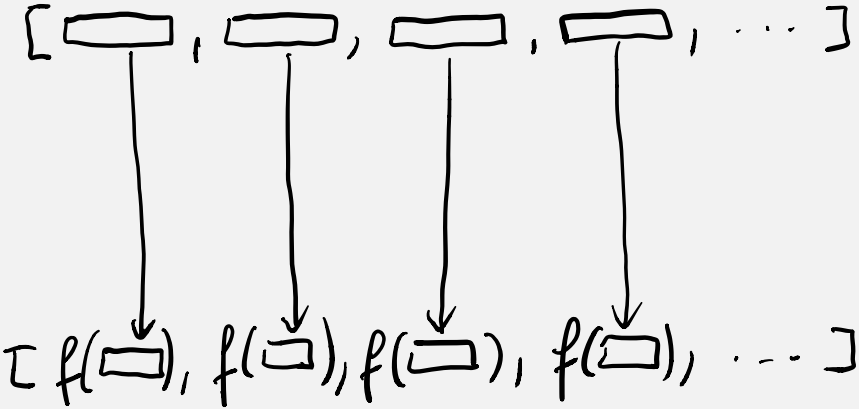
In [4]:
```
def applyFunction(fun, a, b):
    return fun(a, b)

applyFunction(makeList, 1, 2)
```

Out[4]:
```
[1, 2]
```

UNIVERSITÄT BIELEFELD

Map

map(f, collection)



returns

<footer>Programming (Advanced Programming): Functional Programming 7</footer>

Mapping

Mapping is a very powerful concept:

In [5]:
```
def myMapper(f, collection):
    res = list()
    for el in collection:
        res.append(f(el))
    return res

myMapper(str, [1, 2, 3, 4, 5])
```

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

A `map` function is already implemented in Python:

In [6]:
```
map(str, range(1, 6))
```

Out[6]:
```
<map at 0x10620d210>
```

In [7]:
```
list(map(str, range(1, 6)))
```

Out[7]:
```
['1', '2', '3', '4', '5']
```

An example how `map` can be used in practice:

In [8]:
```
import numpy as np

ary = np.random.random(3)
ary
```

Out[8]:
```
array([0.92042043, 0.02694656, 0.8786    ])
```

In [9]:
```
list(map(str, ary))
```

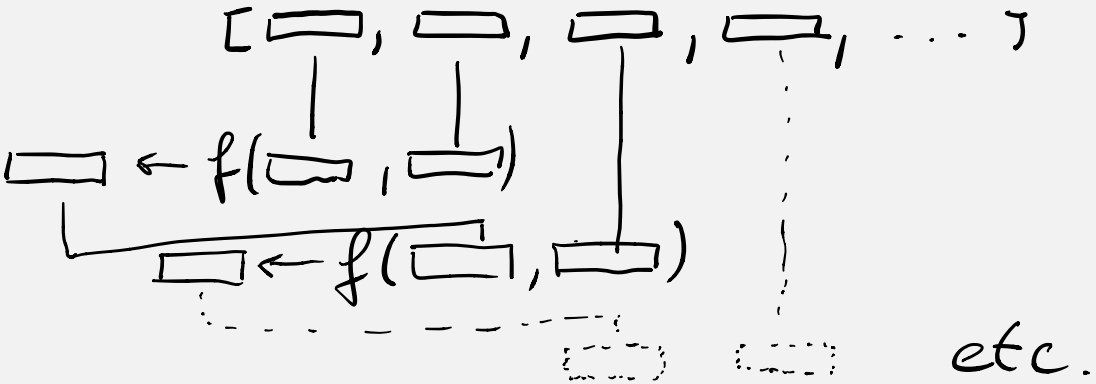
Out[9]:
```
['0.9204204259735078', '0.026946558375632645', '0.8786000025464282']
```

Reduce

reduce(f, collection)

↑ f(x, y)

Iterative reduction of collection through f



Outcome of the last application of f will be returned

Reducing

Another very powerful concept:

In [10]:
```
from functools import reduce

def addition(a, b):
    return a + b

reduce(addition, range(10))
```

Out[10]:
```
45
```

Lambda functions

Lambda functions is just a very convenient way of defining a function in a single line:

In [11]:
```
myFunction = lambda x: f'this value is {x}'

list(map(myFunction, range(3)))
```

Out[11]:
```
['this value is 0', 'this value is 1', 'this value is 2']
```

Here are practical examples, where lambda functions are useful:

In [12]:
```
ary = np.random.random(7)
ary
```

Out[12]:
```
array([0.24786278, 0.13696996, 0.78356076, 0.84696087, 0.7122092 ,
       0.08724698, 0.38522756])
```

In [13]:
```
list(map(lambda x: f'{x:.2f}', ary))
```

Out[13]:
```
['0.25', '0.14', '0.78', '0.85', '0.71', '0.09', '0.39']
```

In [14]:
```
reduce(lambda x, y: x + y, range(10))
```

Out[14]:
```
45
```

Lambda functions can contain conditional statements:

In [15]:
```
myFun = lambda x: x > 10 and 'larger 10' or 'smaller 10'

list(map(myFun, range(8, 13)))
```

Out[15]:
```
['smaller 10', 'smaller 10', 'smaller 10', 'larger 10', 'larger 10']
```

In [16]:
```
list(filter(lambda x: x > 10, range(8, 13)))
```

Out[16]:
```
[11, 12]
```

In [17]:
```
lst = [(4, 'a'), (1, 'a'), ('3', 'c'), (1, 'b'), (2, 'd'), (3, 'e')]

sorted(lst, key = lambda x: x[1])
```

Out[17]:
```
[(4, 'a'), (1, 'a'), (1, 'b'), ('3', 'c'), (2, 'd'), (3, 'e')]
```

List comprehension

In [18]:
```
[x for x in range(10)]
```

Out[18]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List comprehension with conditional (filter) statement:

In [19]:
```
my_list = [1, 'c', 1.0, 'hello world', 'a', 2, 4, 'b', 3.9]

[x for x in my_list if type(x) == str]
```

Out[19]:
```
['c', 'hello world', 'a', 'b']
```

List comprehension where control variable is further modified prior to output:

In [20]:
```
[[x]  for x in my_list if type(x) == str]
```

Out[20]:
```
[['c'], ['hello world'], ['a'], ['b']]
```

Quiz

- *True* or *false?*
  - The `map` functions applies a function to each element of a collection
  - The `filter` function discards all elements which satisfy a given condition
  - The `reduce` function reduces a collection by summing up its elements
  - List comprehensions enable easy list constructions in one line of code
- What is the result of the follwoing expressions?
  - `list(map(lambda x: x*x-x, range(5)))`
  - `list(filter(lambda x: x%2 == 1, range(10)))`
  - `reduce(lambda x,y: x-y, range(5,0,-1))`
  - `[x-1 for x in range(10)if x%2 == 1]`

Quiz

- *True* or *false?*
  - The `map` functions applies a function to each element of a collection — true
  - The `filter` function discards all elements which satisfy a given condition — false
  - The `reduce` function reduces a collection by summing up its elements — false
  - List comprehensions enable easy list constructions in one line of code — true
- What is the result of the follwoing expressions?
  - `list(map(lambda x: x*x-x, range(5)))` — `[0, 0, 2, 6, 12]`
  - `list(filter(lambda x: x%2 == 1, range(10)))` — `[1, 3, 5, 7, 9]`
  - `reduce(lambda x,y: x-y, range(5,0,-1))` — `-5`
  - `[x-1 for x in range(10)if x%2 == 1]` — `[0, 2, 4, 6, 8]`

Functional Programming

Lazy Evaluation

Object-oriented Programming

Lazy Evaluation

Lazy evaluation means that code statements are not executed until their results are really needed.

List comprehensions are turned into generators by using the round brackets.

In [21]:
```
([x]  for x in my_list if type(x) == str)
```

Out[21]:
```
<generator object <genexpr> at 0x10a1a2a50>
```

Using the `next()` function, the currently iterated element of a generator can be retrieved.

In [22]:
```
my_gen = ([x]  for x in my_list if type(x) == str)

next(my_gen)
```

Out[22]:
```
['c']
```

In each call of `next()`, a generator advances its pointer to the current element and returns it, unless the last element has been already reached. In that case, a `StopIteration` exception is thrown.

In [23]:
```
print(next(my_gen))
print(next(my_gen))
print(next(my_gen))
```

```
['hello world']
['a']
['b']
```

In [24]:
```
my_gen = ([x]  for x in my_list if type(x) == str)

try:
    while True:
        print(next(my_gen))
except StopIteration:
    pass
```

```
['c']
['hello world']
['a']
['b']
```

Python uses lazy evaluation wherever possible:

In [25]:
```
rng = range(1, 6)
print(rng)

map_generator = map(str, rng)
print(map_generator)
```

```
range(1, 6)
<map object at 0x10a1a4c10>
```

In [26]:
```
from itertools import repeat

map(str, repeat(1))
```

Out[26]:
```
<map at 0x107efc550>
```

You can create own generator using the `yield` command to return intermediate results in a function.

In [27]:
```
def myMapper(fun, collection):
    for el in collection:
        yield fun(el)
    return res

my_gen = myMapper(str, repeat(1))
my_gen
```

Out[27]:
```
<generator object myMapper at 0x10625c450>
```

To demonstrate the power of lazy evaluation, a mapping has been applied to an infinite sequence (`repeat(1)`). You can obtain the first `x` elements of a sequence using another generator function called `islice` :

In [28]:
```
from itertools import islice

list(islice(my_gen, 10))
```

Out[28]:
```
['1', '1', '1', '1', '1', '1', '1', '1', '1', '1']
```

It is important to understand that generators can only be iterated over once:

In [29]:
```
map_generator = map(str, range(1, 6))
list(map_generator)
```

Out[29]:
```
['1', '2', '3', '4', '5']
```

In [30]:
```
list(map_generator)
```

Out[30]:
```
[]
```

Quiz

- *True* or *false?*
  - Lazy Evaluation makes it possible to retrieve elements from an infinite sequence
  - Generators can be iterated over an arbitrary number of times
  - `range`, `map` and `filter` return generators

Quiz

- *True* or *false?*
  - Lazy Evaluation makes it possible to retrieve elements from an infinite sequence — true
  - Generators can be iterated over an arbitrary number of times — false
  - `range`, `map` and `filter` return generators — true

Summary

- Functional programming
  - Map, reduce, …
  - Lambda functions
  - List comprehension
- Lazy evaluation
  - Generators and iterators