Few words

So you made it! Even though you will go through the following lessons at home, don't hesitate to contact us if there's anything unclear.

The following lessons are quite advanced, but don't let this discourage you. Take your time, play with the code and you will be just fine :)

Let's get to it!

Overview

In this lesson, we will go deeper into iterations and learn how to further optimize you code with:

- Iteration protocal
- Comprehensions
- Anonymous Functions
- Functional Programming Tools

Wikipedia Links

PYTHON ACAD… / 13. [HOME] ITERATION PROTOCOL & COMPREHE… / REVIEW EXERCISES - SCRA… / WIKIPEDIA LI…

Your task is to create a short script that will collect all **the links and the link text** from the main Wikipedia page:

Acquired data should be written into a csv file in form:

```
|Link Text|Link URL|
```

Example of a extract from the resulting csv, could look like this:

```
...
decompression sickness,/wiki/Decompression_sickness
living in a pressurized environment,/wiki/Saturation_diving
Atmospheric diving suits,/wiki/Atmospheric_diving_suits
breath-hold diving,/wiki/Breath-hold_diving
```

```
6 ...
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code. Click to see our solution

Click to see our solution

Wikipedia Page Headers

PYTHON ACA... / 13. [HOME] ITERATION PROTOCOL & COMPRE... / REVIEW EXERCISES - SC... / WIKIPEDIA PAGE HE...

Your task is to create a short script that will collect all **the headers** from the main Wikipedia page.

The aim is to print the layout of the page showing only its headers. Headers on different levels should be indented using `'.'` . Each new level should be indented 4 dots to the right:

- **h1** headers should not be indented
- **h2** headers indented by 4 dots
- **h3** headers indented by 8 dots, etc.

```
$ python print_headers.py
Main Page
....From today's featured article
....Did you know...
....In the news
....On this day...
....Today's featured picture
....Other areas of Wikipedia
....Wikipedia's sister projects
....Wikipedia languages
....Navigation menu
........Personal tools
```

```
........Namespaces
........Variants
........Views
........More
........Search
........Navigation
........Interaction
........Tools
........Print/export
........In other projects
........Languages
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code. Click to see our solution

Click to see our solution

Football Results

PYTHON ACAD… / 13. [HOME] ITERATION PROTOCOL & COMPREHE… / REVIEW EXERCISES - SCR… / FOOTBALL RES…

Create a script that will collect all the football match results from the BBC page.

We would like to scrape all the results for the 1st of October 2017.

The collected data should be stores in a JSON format in a file `scores.json` .

Example of how the data could be collected:

```
{
    "2017-10-01": [
        {
            "Premier League": [
```

```
            {
                "Brighton & Hove Albion": "0",
                "Arsenal": "2"
            },
            {
                "Everton": "0",
                "Burnley": "1"
            },
            {
                "Newcastle United": "1",
                "Liverpool": "1"
            }
        ]
    },
    {
        "Championship": [
            {
                "Sheffield Wednesday": "3",
                "Leeds United": "0"
            }
        ]
    },
    ....
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code. Click to see our solution

Click to see our solution

URL opener

Your task is to create a factory function that will serve as a generator of URL opener functions. That means that the wrapper function takes one argument in form of URL of a resource, we would like to request. The inner function should then collect the optional parameters which can further refine our request.

An example would be a request for overview of transparent account at `"https://www.fio.cz/ib2/transparent"`

Example of use:

```
>>> fio = opener("https://www.fio.cz/ib2/transparent")
>>> f_date = dt(2017,9,1)
>>> t_date= dt.today()
>>> template = '%d.%m.%Y'
>>> f = f_date.strftime(template)
>>> t = t_date.strftime(template)
>>> r = fio(a=2501277007,f=f,t=t)
>>> r.status_code
200
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

What is a protocol?

Protocol prescribes the set of steps that have to be followed in order we achieve our goal. Diplomatic protocol describes how state officials should behave when visiting other country's representatives.

And the same is valid for iteration protocol. It describes the set of steps that are performed, in order Python program can walk over a collection of items.

Iteration protocol is used by `for loop` for example:

```
for letter in 'Hello':
    print(letter)
```

There is a set of steps that Python performs behind the scenes, when executing the above code. Python knows, what steps to perform, because it knows the iteration protocol.

There are 2 important terms to be defined, before we disect the iteration protocol into individual steps:

- **iterable**
- **iterator**

Iterable Object

Iterable object is such an object, that **can be walked through** - usually an object that serves as a **container of other objects**.

Iterable object can be passed into the built-in function iter(), which **accepts only iterable objects** as arguments and returns so called iterator object.

Note that numeric data types are not iterable, therefore we get `TypeError` :

```
>>> iter(25)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

Strings are iterables and therefore no error is raised, when passed into `iter()` :

```
>>> iter('25')
<str_iterator object at 0x7f1c86130550>
```

Iterator Object

It is important to realize, that not the **iterable object** but the **iterator object** is actually walked through during the for loop.

Python creates the iterator object behind the scenes at the beginning of the for loop. Once we have the iterator object, we can loop over it. Looping is performed by repeatedly calling the built-in function next() with the iterator object as argument:

```
>>> iterator_obj = iter('Hello')
>>> next(iterator_obj)
'H'
>>> next(iterator_obj)
```

```
'e'
>>> next(iterator_obj)
'l'
```

Function `next()` advances to the next item of iterator object each time we call it on that object. Once we reach the end of the iterated sequence `StopIteration` exception is raised:

```
>>> next(iterator_obj)
'l'
>>> next(iterator_obj)
'o'
>>> next(iterator_obj)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Iteration Protocol

Iteration protocol is described by the following set of steps:

1. Take **iterable** object and pass it to the `iter()` function
2. `iter()` function returns **iterator** object
3. Store the **iterator** object in a variable
4. Repeatedly pass the **iterator** object into the `next()` function
5. `next()` function returns the next item from the iterator object on each call
6. Once all the items have been returned from the iterator object, the `StopIteration` exception is raised to tell us, that we arrived at the end of the sequence

The goal of the iteration protocol is thus to retrieve all the items from the container, one by one.

Example of:

```
>>> iterator = iter('abc')
```

```
>>> next(iterator)
'a'
>>> next(iterator)
'b'
>>> next(iterator)
'c'
>>> next(iterator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

It is important to understand that iterable objects cannot be passed directly into the `next()` function (that actually retrieves the individual items)

```
>>> next('Hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object is not an iterator
```

Iterable Data Types

Iterable data types are those that support use of function `iter()` . This function creates **iterator object** from an iterable object:

Sequence data types are for example iterable, so let's demonstrate the working of `iter()` function on them:

```
>>> lst = [1,2,3]
>>> iterator = iter(lst)
>>> iterator
<list_iterator object at 0x7f67ac6dc278>
>>> type(lst) == type(iterator)
False
```

We can see that the iterator object is a new type of object.

It is important to note that if we pass to `iter()` an **iterator object**, the same **iterator object** is returned.

```
>>> iterator2 = iter(iterator)
>>> iterator2 is iterator
True
```

Examples of iterable objects

1. Sequences (str, list, etc.)

```
>>> for i in range(2):
...     print(i)
0
1
```

2. Dictionary and dictionary views

```
>>> d
{'surname': 'Smith', 'name': 'John'}
>>> for key in d:
...     print(k)
...
dict_keys(['surname', 'name'])
dict_keys(['surname', 'name'])
```

```
>>> for value in d.values():
...     print(value)
...
Smith
John
```

3. Sets

```
>>> s
{'surname', 'name'}
>>> for i in s:
```

```
...     print(i)
...
surname
name
```

4. Files

```
>>> f = open('test.txt')
>>> for line in f:
...     print(line)
...
First line
Second line
```

Important Additional details

PYTHON ACAD… / 13. [HOME] ITERATION PROTOCOL & COMPRE… / ITERATION PROT… / IMPORTANT ADDITIONAL D…

It is important to point out, that some objects are already iterators and therefore they do not have to be passed into the `iter()` function before sent to `next()`. An example would be the object returned by the built-in `reversed()` function:

```
>>> r = reversed('Hello')
>>> r
<reversed object at 0x7f1c861306a0>
>>> next(r)
'o'
>>> next(r)
'l'
>>> next(r)
'l'
>>> next(r)
'e'
>>> next(r)
'H'
```

Also file objects are iterators, that do not have to be passed into the `next()` function:

```
>>> f = open('test.txt')
>>> next(f)
'First line\n'
>>> next(f)
'Second line'
>>> next(f)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

If we pass the iterator object into the `iter()` function, the same object is returned back. We can prove that by

- starting calling `next()` on file object,
- then passing it into `iter()` and store the output in a new variable (`f2`)
- then call `next()` on the new variable.

Iterator object **keeps the counter of the next item to be returned** and so the second line (not the first line) from the file is printed.

```
>>> next(f)
'First line\n'
>>> f2 = iter(f)
>>> next(f2)
'Second line'
>>> next(f2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

For Simulated by While

PYTHON ACAD… / 13. [HOME] ITERATION PROTOCOL & COMPREHE… / ITERATION PROTO… / FOR SIMULATED BY W…

Knowing, how iteration protocol works, we can simulate it using `while` loop. Let's say we would like to translate the following for loop into a while loop:

```
for num in range(5):
    print(num)
```

```
rng_iterator = iter(range(5))
while True:
    try:
        num = next(rng_iterator)
        print(num)
    except StopIteration:
        break
```

Now you can see, why `for` loop is such a great thing - it is able to reduce 7 lines of code into 2 lines.

Why Learn This Protocol

PYTHON ACAD… / 13. [HOME] ITERATION PROTOCOL & COMPREHE… / ITERATION PROTO… / WHY LEARN THIS PROT…

Besides better understanding how things work under the hood, iteration protocol is heavily used with so called generator objects. We will learn about these in one of the upcoming lessons. Generators can incredibly speed up execution of our programs.

So knowing iteration protocol can make our programs much faster in the near future.

What is a comprehension?

Is an expression, that allows for building of a list, dictionary or set object with one line of code. It does in one line does what a following for loop does in three lines

```
>>> my_list = []
>>> for i in range(5):
...     my_list.append(i)
...
>>> my_list
[0, 1, 2, 3, 4]
```

An equivalent to the above code generating a new list would be a list comprehension:

```
>>> [num for num in range(5)]
[0, 1, 2, 3, 4]
```

It is good to note that list comprehension is faster because it has been optimized for the Python interpreter which can spot a predictable pattern during looping.

So let's speak about the comprehension syntax details.

List Comprehension Syntax

We can disect the most basic comprehension expression as follows (below we have an example of **list comprehension**:

```
1 [          expression        for item in iterable           ]
2    |___|       |__________|      |____________________|
   |___|
3 opening bracket appended item     for loop generating items
  closing bracket
```

Above we have depicted the structure of list comprehension that is:

1. **enclosed** in square brackets, because it generates a list.
2. opening bracket is followed by a variable name - or more generally an **expression** - that represents the item that is being stored on each iteration
3. the fact that the iteration is going on here is implied by the **for loop header**
4. the whole expression is terminated by the **closing bracket**

So to generate a list of numbers, we could do it as follows:

```
>>> [num for num in range(5)]
[0, 1, 2, 3, 4]
```

The **expression** following the square brackets **can be more sophisticated**. Below we generate a new list of uppercased strings:

```
>>> names = ['bob', 'john', 'frank']
>>> upper_names = [name.upper() for name in names]
>>> upper_names
['BOB', 'JOHN', 'FRANK']
```

Set Comprehension Syntax

Set comprehension generates a set object in one line. The syntax is almost identical to that of list comprehension with the difference of enclosing the expression in curly braces:

```
{           expression        for item in iterable           }
   |___|        |_________|      |____________________|
   |___|
opening bracket appended item     for loop generating items
closing bracket
```

So the example would be:

```
>>> nums = [68, 48, 38, 29, 68, 29, 72, 86, 25, 92]
>>> unique_nums = {num for num in nums}
>>> unique_nums
{68, 38, 72, 48, 86, 25, 92, 29}
```

Dictionary Comprehension Syntax

Dictionary comprehension generates a dictionary in one line of code. It can be identified by being **enclosed inside curly braces** and **key:value pairs** at the beginning of comprehension expression (following opening curly brace).

The syntax is pretty similar to that of list and mainly set comprehension with the difference that we need to generate **key:value** pairs to make it clear that a dictionary is being built:

```
{           key:value         for key,value in iterable
}
   |___|        |_________|      |_____________________________|
   |___|
opening bracket  appended k:v       for loop generating items
closing bracket
```

We can of course apply dict comprehension to `dict.items()` or an iterable that consists of 2-item subsequences (f.e. tuple `(10,5)`).

Example:

```
>>> people = (('John',23),('Bob',43),('Fred',54))
>>> people_dict = {name:age for name, age in people}
>>> people_dict
{'John': 23, 'Fred': 54, 'Bob': 43}
```

Conditions in Comprehensions

Comprehensions can also contain conditions, that append generated item to the structure being built only if the condition evaluates `True`.

General syntax would be:

```
[            item            for item in iterable     if test
]
   |___|         |____|         |____________________|   |______|
|___|
opening bracket   appended item  for loop generating items  condition
closing bracket
```

Example of comprehension that filters only even numbers from a range:

```
>>> even = [num for num in range(15) if num%2==0]
>>> even
[0, 2, 4, 6, 8, 10, 12, 14]
```

Having the condition part in a comprehension means, that only items, that meet the given condition will be filtered out into the newly created collection.

There can be multiple if statements at the end of the comprehension:

```
>>> [num for num in range(10) if num > 5 if num %2==0]
[6, 8]
```

But this has the same effect, as using logical `and` operator in one conditional statement:

```
>>> [num for num in range(10) if num > 5 and num %2==0]
[6, 8]
```

Multiple For Headers in a Comprehension

Sometimes, in our code, we use nested for loops. For example in the code below we are trying to join letters at the same positions in two strings:

```
>>> my_list = []
>>> for x in 'abc':
...     for y in 'def':
...             my_list.append((x,y))
...
```

We can write the above nested for loops in one comprehension by chaining multiple `for` headers in it:

```
>>> nums = [(x,y) for x in 'abc' for y in 'def']
```

This is the pretty printed result:

```
>>> from pprint import pprint as pp
>>> pp(nums)
[('a', 'd'),
 ('a', 'e'),
 ('a', 'f'),
 ('b', 'd'),
 ('b', 'e'),
 ('b', 'f'),
```

```
 ('c', 'd'),
 ('c', 'e'),
 ('c', 'f')]
```

The first for loop in the comprehension is the first one in the nested for loop and the second for structure represents the nested for loop in the example above.

Of course we could create even more complex comprehension with more than 2 loops, however, the readbility of such expression would decrease gradually.

Nested Sequences with Comprehensions

If we would like to create structures inside structures, we will need to nest comprehensions inside each other. Example is a list of lists, where the inner lists can represent rows in a table:

```
>>> [['#' for col in range(5)] for row in range(5)]
[['#', '#', '#', '#', '#'], ['#', '#', '#', '#', '#'], ['#', '#', '#',
'#', '#'], ['#', '#', '#', '#', '#'], ['#', '#', '#', '#', '#']]
``
```

Pretty printing the nested sequence will show its structure better:

```
>>> import pprint
>>> pprint.pprint([['#' for col in range(5)] for row in range(5)])
[['#', '#', '#', '#', '#'],
 ['#', '#', '#', '#', '#'],
 ['#', '#', '#', '#', '#'],
 ['#', '#', '#', '#', '#'],
 ['#', '#', '#', '#', '#']]
```

What is Anonymous Function

We have learned that functions are created using `def` statement, that are composed of header and body (suite). The header part contains the function's name.

In this section we will learn about functions that have **no name** and are created using **one-line expression**. These functions are called **lambda** or **anonymous** functions.

We create lambda functions using `lambda` keyword, followed by **one or more arguments** separated by **colon** from the code executed by the function.

```
lambda argument1, arg2, ... : one-line expression which result is returned
```

Example of anonymous function definition:

```
lambda arg: arg+2
```

We do not write `return` keyword inside the `lambda` expression.

Lambda Function vs. Traditional Function

PYTHON AC… / 13. [HOME] ITERATION PROTOCOL & CO… / ANONYMOUS FU… / LAMBDA FUNCTION VS. TRADITIO…

To understand lambda function definition better, let's try to convert it to traditional function definition.

We have the following anonymous function:

```
lambda arg: arg+2
```

This function takes in 1 argument `arg` and returns the result of expression `arg+2` . This can be tranlated into traditional function definition as follows:

```
def add_two(arg):
    return arg+2
```

The difference is that we had to include function name and `return` keyword in traditional definition.

Where to Use Lambda Functions

PYTHON ACA… / 13. [HOME] ITERATION PROTOCOL & COMPR… / ANONYMOUS FUNC… / WHERE TO USE LAMBDA FU…

Lambda functions are usually passed as arguments into another function calls. Example could be function `sorted()` :

```
>>> sorted('Good News Today', key= lambda x: x.upper())
[' ', ' ', 'a', 'd', 'd', 'e', 'G', 'N', 'o', 'o', 'o', 's', 'T', 'w',
'y']
```

The function refered by the `key` parameter is applied to each item of the sequence. Actually the result returned by that lambda function is then taken into consideration, when sorting items in the sequence. Every letter was converted to uppercase and then sorted according to the ASCII value of its uppercase equivalent.

You can see the difference, in the result, if no `key` argument is provided - letters are ordered according to their actual ASCII code value:

```
>>> sorted('Good News Today')
[' ', ' ', 'G', 'N', 'T', 'a', 'd', 'd', 'e', 'o', 'o', 'o', 's', 'w',
'y']
```

List data type has also `sort()` built-in method, that can also be passed lambda function to the `key` parameter:

```
>>> lst = list('Good News Today')
>>> lst
['G', 'o', 'o', 'd', ' ', 'N', 'e', 'w', 's', ' ', 'T', 'o', 'd', 'a',
'y']
>>> lst.sort(key=lambda x: x.upper())
>>> lst
[' ', ' ', 'a', 'd', 'd', 'e', 'G', 'N', 'o', 'o', 'o', 's', 'T', 'w',
'y']
```

Or even `max()` and `min()` functions accept `key` parameter. In the example below, we extract the longest string the in the list `names`. In other words, we extract an item, that if passed to the function `lambda x: len(x)`, this function will return the highest value:

```
>>> names = ['Bob','Nick', 'Ann', 'Johnny', 'Elisabeth']
>>> max(names, key= lambda x: len(x))
'Elisabeth'
```

Lambda functions are also often combined with functional programming tools as `map()`, `filter()`, `reduce()`.

```
>>> list(map(lambda x: x.upper(),'abcdefgh'))
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
```

We will learn about functional programming tools in the next section.

Anonymous Function with Name

It does not mean that if a function is anonymous - it does not have a name - we cannot give it one. Lambda expression returns function object. Once we assign this object to a variable, we can call it as a traditional named function:

```
>>> add_two = lambda arg: arg + 2
>>> add_two(2)
4
```

Lambdas and Scope

Variables assigned within lambda function (its arguments and variables assigned in the body) also pertain to the lambda function's local scope.

```
>>> lst = ['a','b','c']
>>> lambda lst: lst.append('d')
<function <lambda> at 0x7fb162437c80>
>>> lst
['a', 'b', 'c']
```

Intro

In this section we will learn about three functions:

1. `map()`
2. `filter()`
3. `reduce()`

Each of them does something different, but one thing they have in common, they take another function and apply it accross items in a container. All of these functions make use of **iteration protocol** to step through the whole collection of items in a container object - sequences, dictionary views, sets.

Example:

Below applyt string method `upper()` to all items in the string `'abcde'`

```
>>> list(map(str.upper,'abcde'))
['A', 'B', 'C', 'D', 'E']
```

We had to convert the result returned by the `map()` function into a list. Why is this so, we will explain in the next section.

So the **syntax** is always:

```
functional_tool(function, iterable)
```

What Objects are Returned?

PYTHON AC… / 13. [HOME] ITERATION PROTOCOL & COM… / FUNCTIONAL PROGRAMMI… / WHAT OBJECTS ARE RE…

Values returned by the functions `map()` and `filter()` may seem strange at the first sight:

```
>>> map(str.upper, 'abcde')
<map object at 0x7fb1608e92e8>
```

```
>>> filter(lambda x: x%2==0, range(10))
<filter object at 0x7fb1608e9358>
```

In order to see the contents of the `map` of `filter` objects, we need to pass them into the `list()` function:

```
>>> list(map(str.upper, 'abcde'))
['A', 'B', 'C', 'D', 'E']
>>> list(filter(lambda x: x%2==0, range(10)))
[0, 2, 4, 6, 8]
```

Therefore, please, be not surprised we will convert these objects into lists to demonstrate the results they produce.

map() Function

PYTHON ACAD… / 13. [HOME] ITERATION PROTOCOL & COMPREH… / FUNCTIONAL PROGRAMMING … / MAP() FUNC…

Function `map()` applies a certain function to each item of the iterable.

For instance, in the example below it applies the function `upper()` to each letter of the string `abcde` :

```
>>> map(str.upper,'abcde')
<map object at 0x7f7d94eb6668>
```

If we want to see the content refered by the map object, we need to convert it into a list of tuple:

```
>>> list(map(str.upper,'abcde'))
['A', 'B', 'C', 'D', 'E']
```

We can define our own function - for example `increase_by_1()` :

```
def increase_by_1(num):
    return num + 1
```

And use this function inside `map()` :

```
>>> lst = [1, 2, 3, 4]
>>> list(map(increase_by_1,lst))
[2, 3, 4, 5]
```

Function `map()` applied `increase_by_1()` to each member of the list `lst` .

map() Behind the Scenes

PYTHON ACA… / 13. [HOME] ITERATION PROTOCOL & COMP… / FUNCTIONAL PROGRAMMIN… / MAP() BEHIND THE …

What `map()` does behind the scenes is it uses **iteration protocol**. Therefore we can simulate its inner workings through a `for` loop.

So when we have a function `increase_by_1()` :

```
>>> def increase_by_1(num):
...     return num + 1
```

Using `map()` , the result is as follows:

```
>>> lst = [1, 2, 3, 4]
>>> list(map(increase_by_1,lst))
[2, 3, 4, 5]
```

What `map()` actually does can be translated into the following `for` loop:

```
result = []
lst = [1,2,3,4]
for item in lst:
    result.append(increase_by_1(item))
return result
```

Advanced map()

An advanced use of `map()` is to pass it **multiple iterables**:

```
map(func, iterable1, iterable2, ...)
```

The number of iterrables (e.g. lists, tuples, etc.) has to correspond to the number of arguments the `func` argument expects.

Let's say, we have defined our own function `sum_two()`:

```
>>> def sum_two(a,b):
...     return a+b
```

This function expects two arguments. Therefore, if applied using `map()`, also the number of iterables passed to `map()`, has to be equal to two:

```
>>> lst1 = [1,2,3,4]
>>> lst2 = [5,6,7]
>>> list(map(sum_two, lst1, lst2))
[6, 8, 10]
```

We can see that `map()` iterates over `lst1` and `lst2` and sends items at corresponding indices to the function `sum_two()`. The result is a sequence of sums:

- 1 + 5 = 6
- 2 + 6 = 8

- 3 + 7 = 10

Lambda & map()

PYTHON ACAD… / 13. [HOME] ITERATION PROTOCOL & COMPREH… / FUNCTIONAL PROGRAMMING … / LAMBDA & M…

The `increase_by_1()` example function is so simple, that we could write it as a lambda function directly inside the `map()` :

```
>>> lst = [1,2,3,4]
>>> m = map(lambda x: x+1, lst)
>>> list(m)
[2,3,4,5]
```

filter() Function

PYTHON ACAD… / 13. [HOME] ITERATION PROTOCOL & COMPRE… / FUNCTIONAL PROGRAMMING … / FILTER() FUNC…

Similarly to `map()` function, `filter()` applies a function to every item of a given iterable. This time, however, the result returned by `filter()` contains only those values, for which function returns `True` . It effectively filters those values out of a given iterable.

We can define our own function `func()` and the returned value will be checked for its boolean value.

```
>>> def func(char):
...     if char in 'aeiouy':
...             return char
...     else:
...             return ''
>>> list(filter(func,'abcdef'))
['a', 'e']
```

So for the above string `abcdef`, the function `func` returns the following sequnce of values: `'a','', '', '', 'e',''`. Function `filter()` then evaluates boolean value of each of those results and keeps, only those, that return `True` (`'a','e'`).

filter() Behind the Scenes

We could simulate filter function using for loop and list to collect the results.

Filtering only uppercase letters from `'AbCdEf'` with `filter()`:

```
>>> list(filter(str.isupper,'AbCdEf'))
['A', 'C', 'E']
```

Doing the same with **for loop**:

```
>>> my_str = 'AbCdEf'
>>> result = []
>>> for char in my_str:
...     if char.isupper():
...             result.append(char)
...
>>> result
['A', 'C', 'E']
```

As we can see, filter allows for less lines of code than the for loop in this case.

To make the for loop code shorther we can use **list comprehension**:

```
>>> result = [char for char in my_str if char.isupper()]
>>> result
['A', 'C', 'E']
```

Lambda & filter()

We can easily filter integers greater than 5 from `range(-5,10)` , using **lambda function**:

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

The important part is, that `x>5` already returns a boolean value.

reduce() Function

Function `reduce()` is defined inside functools module and in order to access it, we need to import it first:

```
from functools import reduce
```

This function has an extra parameter:

```
reduce(func, iterable [,initializer])
```

This function does not return iterable object as filter or map. The original iterable passed in, is **reduced to a single values**. It applies a function of two arguments cumulatively to the items of a sequence, from left to right, so as to reduce the sequence to a single value.

For example factorial 5 (5 x 4 x 3 x 2 x 1), can be written as follows:

```
>>> reduce(lambda x,y:x*y,range(1,6))
120
```

You can visualize the code with Python Tutor.

The optional initializer argument provides a default value and is placed before the items of the sequence in the calculation:

```
>>> reduce(lambda x,y: x+y, range(1,5),5)
15
```

What is actually happening above is the summation `((((5 + 1)+2)+3)+4)` with number 5 at the beginning of the calculation. If the given sequence is empty, the initializer value is returned:

```
>>> reduce(lambda x,y: x+y, range(1,1),5)
5
```

reduce() Behind the Scenes

PYTHON ACA… / 13. [HOME] ITERATION PROTOCOL & COM… / FUNCTIONAL PROGRAMMI… / REDUCE() BEHIND THE…

The `reduce()` code below:

```
>>> from functools import reduce
>>> reduce(lambda x,y:x*y,range(5,1,-1))
120
```

… can be rewritten by using for loop (not comprehension - comprehensions build a collection, meanwhile reduce returns a single value):

```
>>> result = 1
>>> for num in range(5,1,-1):
...     result *= num
...
>>> result
120
```

Filter & Map Object

PYTHON ACA… / 13. [HOME] ITERATION PROTOCOL & COMPRE… / FUNCTIONAL PROGRAMMIN… / FILTER & MAP O…

Neither map nor filter objects support indexing, even though they support iteration.

```
>>> lst = [1,2,3,4]
>>> m = map(lambda x: x+1, lst)
>>> m[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'map' object is not subscriptable
```

Another important characteristic of filter and map objects is that we can iterate them only once:

```
>>> m = map(lambda x: x+1, lst)
>>> for item in m:
...     print(item,end=', ')
...
2, 3, 4, 5,
>>> for item in m:
...     print(item, end=', ')
...
>>> list(m)
[]
```

The `list()` function as well as `for` loop, both of them use iteration protocol. However, once the `map()` object has been iterated, it does not yield any more values.

```
>>> lst = [1,2,3,4]
>>> m = map(lambda x: x+1, lst)
>>> next(m)
2
>>> next(m)
3
>>> next(m)
4
>>> next(m)
5
>>> next(m)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Any further call to `next()` function would cause `StopIteration` exception

Why operator Module

PYTHON ACADE… / 13. [HOME] ITERATION PROTOCOL & COMPREHEN… / MODULE OPERAT… / WHY OPERATOR MOD…

We introduce the `operator` module here, because it is often useful in combination with functions `map()`, `filter()` and `reduce()`. This module represents (exports) all the Python operators in form of functions.

So for example, if we wanted to **add** two numbers, but avoid using `+` operator, we can do it as follows:

```
>>> import operator as op
```

```
>>> op.add(2,3)
5
```

To multiply two numbers, we use `operator` 's `mul()` function:

```
>>> op.mul(2,3)
6
```

Operator module's functions can substitute the use of lambda functions. For example the `op.mul()` function can be written as lambda functions:

```
lambda x,y: x*y
```

To learn more functions available in the module, check the module's documentation

map() & operator Module

PYTHON ACAD… / 13. [HOME] ITERATION PROTOCOL & COMPREHE… / MODULE OPERAT… / MAP() & OPERATOR MO…

An example of using `operator` module's function in combination with `map()` could be to convert all the iterable's value into their absolute value:

```
>>> nums = [0, -1, -7, 2, 4, -8, 3, 8, -8, -10]
>>> positive = list(map(op.abs,nums))
>>> positive
[0, 1, 7, 2, 4, 8, 3, 8, 8, 10]
```

Equivalent using list comprehension:

```
>>> positive = [op.abs(num) for num in nums]
>>> positive
[0, 1, 7, 2, 4, 8, 3, 8, 8, 10]
```

reduce() & operator Module

The `reduce()` function is ideal place for use of `operator` module's functions. Instead of creating lambda function that subtracts item `a` from item `b` (`lambda a,b: a-b`), we can use `operator`'s `sub()` function:

```
>>> import operator as op
>>> nums = [0, -1, -7, 2, 4, -8, 3, 8, -8, -10]
>>> reduce(op.sub,nums)
17
```

The equivalent statement using **lambda** function would look as follows:

```
>>> reduce(lambda a,b: a-b, nums)
17
```

Iteration Protocol

PYTHON ACADEMY / 13. [HOME] ITERATION PROTOCOL & COMPREHENSIONS / QUIZ / ITERATION PROTOCOL

Nepodařilo se zobrazit část obsahu. Chyba je na naší straně.

Comprehensions

PYTHON ACADEMY / 13. [HOME] ITERATION PROTOCOL & COMPREHENSIONS / QUIZ / COMPREHENSIONS

1/7

What are the benefits of comprehensions?

A. They are always more readable

B. Faster than for loops

C. They do not generate all the items of a collection at once

Functools

PYTHON ACADEMY / 13. [HOME] ITERATION PROTOCOL & COMPREHENSIONS / QUIZ / FUNCTOOLS

1/7

What will be the result of the following expression?:

```
>>> list(map([1,2,3],[2,3,4]))
```

A. TypeError

B. [12,23,34]

C. [1,2,2,3,3,4]

D. [2,3]

Anonymous Functions

PYTHON ACADEMY / 13. [HOME] ITERATION PROTOCOL & COMPREHENSIONS / QUIZ / ANONYMOUS FUNCTIONS

1/5

What will be stored inside the variable `num` ?:

```
>>> num = lambda x,y: x+y
```

A. The result of expression x+y

B. Parameters x and y

C. Variable object

D. Function object

File Clean Up 2

Create a clean-up script that will move files which has not been open for a given number of days into a folder called **Old_Stuff**.

The script should be launched from the command line and shoult take two command line arguments:

- path to the directory, the use would like to get cleaned up
- integer which tells the script, that files older than that number of days should be moved

The script will probably need to use:

- `os.path.getatime(path)` - returns the time of last access of path. The return value is a number giving the number of seconds since the epoch (see the time module). Raise OSError if the file does not exist or is inaccessible.
- `time.time()` - returns the time in seconds since 1.1.1970 as a floating point number. This function will provide you with the current time. If you would like to see more, check module time documentation

Example of running the program:

```
$ python cleanup2.py "/home/martin/PythonBeginner/Lesson8/TestDir1" 5
Moving: test.txt ...to... Old_Stuff
```

If incorrect inputs provided:

```
$ python cleanup2.py "/home/martin/PythonBeginner/Lesson8/TestDir1" a
<older_than> has to be numeric
USAGE: python cleanup2.py <dir_to_clean_path> <older_than>
```

Insufficient number of inputs:

```
$ python cleanup2.py
USAGE: python cleanup2.py <dir_to_clean_path> <older_than>
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Sum rows comprehension

Create a function that will make use of list comprehension to calculate totals for each row of the following table:

```
table = [['Amount1','Amount2', 'Amount3'],
         [  321,       43,       432],
         [ 3213,       42,       482],
         [  543,        38,      232]]
```

Before we knew list comprehensions, we used `for` loops:

```
row_totals = []
for row in table[1:]:
    row_totals.append(0)
    for col in row:
        row_totals[-1] += col
```

Bonus

Try to create another function, that will use `map()` to accomplish the same job.

Example of use with list comprehension:

```
>>> row_totals(table)
[796, 3737, 813]
```

Example of use with `map()` function:

```
>>> row_totals(table)
<map object at 0x7f67ac6fb198>
>>> list(row_totals(table))
[796, 3737, 813]
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Making own map()

PYTHON ACADEMY / 13. [HOME] ITERATION PROTOCOL & COMPREHENSIONS / EXERCISES / MAKING OWN MAP()

To better understand, how does `map()` function look behind the scenes, your task will be to implement your own version of it. You can call it `my_map()` and it should behave as follows:

```
my_map(func, iterable1, iterable2,..)
```

Incorrect input: Number of iterables != Number of arguments required by func

```
>>> import operator as op
>>> my_map(op.add,[1,2])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 13, in my_map
TypeError: op_add expected 2 arguments, got 1
```

Incorrect input: No iterable passed

```
>>> my_map(op.add,1,2)
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in my_map
  File "<stdin>", line 3, in <listcomp>
TypeError: 'int' object is not iterable
```

Correct input: Same number of items in each iterable

```
>>> my_map(op.add,[1,2],[3,4])
[4, 6]
```

Correct input: Different number of items in each iterable

```
>>> my_map(op.add,[1,2],[3,4,5])
[4, 6]
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Making own filter()

PYTHON ACADEMY / 13. [HOME] ITERATION PROTOCOL & COMPREHENSIONS / EXERCISES / MAKING OWN FILTER()

To better understand, how does `filter()` function look behind the scenes, your task will be to implement your own version of it. You can call it `my_filter()` and it should behave as follows:

```
my_filter(func, iterable)
```

Incorrect input: data type passed (not iterable)

```
>>> my_filter(str.isupper,1234)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
  File "<stdin>", line 3, in my_filter
TypeError: 'int' object is not iterable
```

Correct input:

```
>>> my_filter(str.isupper,'Hello')
['H']
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Extracting Emails (Again)

PYTHON ACADE… / 13. [HOME] ITERATION PROTOCOL & COMPREHENSI… / EXERCIS… / EXTRACTING EMAILS (AGA…

We have already solved a task, with extracting e-mail addresses from a string. This time will be our task to do it by using functions `map()` and `filter()`.

Here we have a testing string, from which we would like to extract all the e-mail domains:

```
1 text = 'This is some abc@email.com/ example@mail.org, text that
  contains efg@mail.cz.'
```

Your task is to create a short script, that will extract all the e-mails using functional programming tools we have learned in the previous lesson.

To identify an e-mail, search for a given string containing a character `'@'`.

Example of script at work:

```
$ python get_domains.py
```

```
['email.com', 'mail.org', 'mail.cz']
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

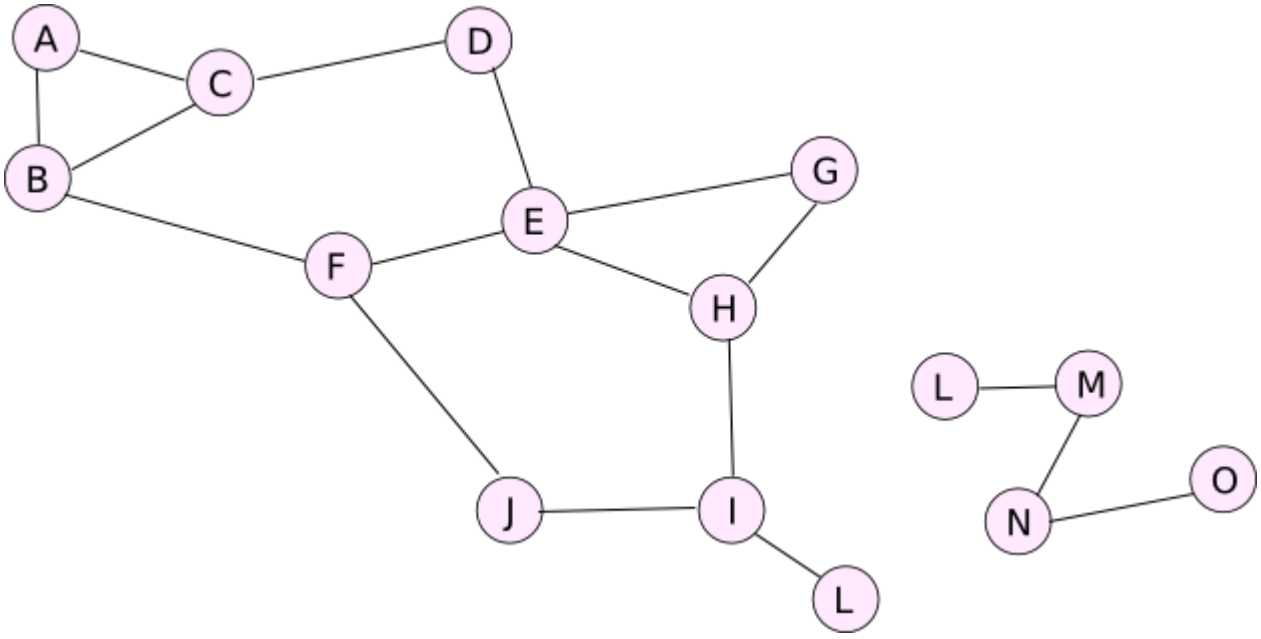
Social Network

PYTHON ACADEMY / 13. [HOME] ITERATION PROTOCOL & COMPREHENSIONS / EXERCISES / SOCIAL NETWORK

We have a social network of people that is mapped to the following stucture:

```
friendships = [('A','B'),
               ('B','C'),
               ('A','C'),
               ('C','D'),
               ('B','F'),
               ('D','E'),
               ('F','E'),
               ('F','J'),
               ('E','H'),
               ('E','G'),
               ('G','H'),
               ('H','I'),
               ('I','J'),
               ('I','L'),
               ('L','M'),
               ('M','N'),
               ('N','O')]
```

Here we have a list of friendships, that is also graphically depicted below:



Would like to know

1. **What friends two given users have in common**. Our task is to create a a function that will list all the friends in common

And this is how our function should work:

```
friends_in_common('A','F') --> only 'B' should be listed
friends_in_common('A','D') --> only 'C' should be listed
```

2. **Whether two given users are connected directly**

```
>>> connected_directly('A','B')
True
>>> connected_directly('A','D')False
False
```

Both functions should be implemented using functional programming tools.

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

DALŠÍ LEKCE