Overview

This is it! Last lesson is here. It's been a ride, huh? If you feel really confident with all the concepts you have learned in this course, you should have very solid base for jumping into the world of Python programming.

If you have any recommendations how we could improve our course (whether this one or any other) do send us your feedback :)

And what will be the subject of this lesson? We're gonna talk about:

- Generators - these are going to upgrade our iteration protocol knowledge to the next level.
- Regular Expressions - concept of regexes is very useful when you're dealing with strings where you aim to find very specific string pattern,
- and Package Importing - here we're gonna built-up on our importing knowledge.

Flatten a nested list

Create a function that will take arbitrarily nested list as a input, and will return a 1-dimensional list. You should test, whether your function works properly by passing different lists.

You can use these test cases:

Input:

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

Expected Output:

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

Input:

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

Expected Output:

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

Input:

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

Expected Output:

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

Example of function in use:

```
>>> lst = [1, [2, 3], [4, [5, 6], 7, [[8, 9]]]]
>>> flatten(lst)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Code Solution

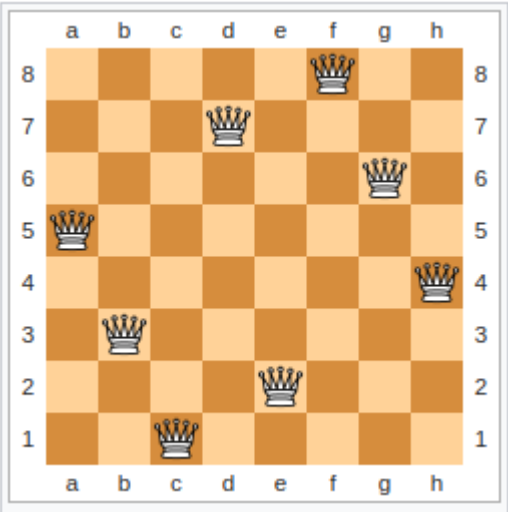
Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

N-Queens Puzzle

Create a function, that will generate all the possible distributions of N queens on N x N chess board. The solution requires that no two queens share the same row, column, or diagonal.

Example of such distribution taken from Wikipedia:



Code Solution

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

Click to see our solution

Time your code

PYTHON ACADE… / 15. [HOME] GENERATORS, REGEX & PACKAGE IMPORTI… / REVIEW EXERCIS… / TIME YOUR CO…

Your task is to create a function called `timer()`, that will take as input another function and its arguments. Inside the `timer()`, the input function should be run. The function should return the number of seconds it took to run the input function.

Example of function in use:

In the code below, we run the function `sum()` 100 times. The function sums numbers between 0 and 35. It took 0.0004 seconds to perform so many repetitions

```
>>> elapsed = timer(sum,range(36),repetitions=100)
>>> elapsed
0.000377655029296875
```

Code Solution

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

Click to see our solution

Intro

We have met so called iterator objects. They bare a very interesting feature, they produce one value at a time. Values from itertors are produced by calling next function as `next(iterator)`. The big advantage of iterator objects is that they save memory resources by generating only one value - the next value in a iterated collection.

Until now, we did not know how do create our own iterator-like objects. But that will no longer be true after reading the section on generators.

Generators are objects that behave like iterators. If passed into a `next()` function call, a value is **yielded** or `StopIteration` error is raised.

Let's look how we can create our own generators.

Generator Functions

Generator functions are special type of Python functions, that differ from the classic functions in one important feature - generator functions contain `yield` statement somewhere in their body.

In the below generator function example we simulate a range object:

```
1  def my_range(start,stop,step=1):
2      test = (lambda x,y: x<y and step>0,
3              lambda x,y: x>y and step < 0)[ start > stop]
4
5      while test(start,stop):
6          yield start
7          start += step
```

And this is how it works:

```
>>> for num in my_range(2,7):
...     print(num, end=' ')
...
```

```
2 3 4 5 6
```

We can generate every nth number:

```
>>> for num in my_range(2,10,2):
...     print(num, end=' ')
...
2 4 6 8
```

We cannot go backwards if `start` is smaller then `stop` :

```
>>> for num in my_range(2,10,-2):
...     print(num, end=' ')
...
>>>
```

These are the correct inputs:

```
>>> for num in my_range(10,2,-2):
...     print(num, end=' ')
...
10 8 6 4
```

In the next section we will dissect the code of our generator function.

Generator Function Dissected

PYTHON ACAD… / 15. [HOME] GENERATORS, REGEX & PACKAGE IMP… / GENERAT… / GENERATOR FUNCTION DISSE…

In the previous section we have created the following generator function and now we will look at its inner workings:

```
1  def my_range(start,stop,step=1):
2      test = (lambda x,y: x<y and step>0, lambda x,y: x>y and step<0)
   [start > stop]
3
4      while test(start,stop):
```

```
        yield start
        start += step
```

On the first row, we choose the correct condition to be tested before each loop iteration. Therefore, if `start` is smaller than `stop`, we want our loop to run until the condition `x (start) > y (stop)` is `True`. The test has to check as well, whether the `step` is correctly negative in this case. This is in order on each iteration, the value of `start` can be decreased. The other lambda function is stored in the `test` variable if `start` is smaller than `stop`.

Loop, that follows **yields** or we can say, produces the value of `start` on each iteration. Subsequently the `start` is change by the value of `step`.

What is important, the code execution stops at the point, when the `start` value is yielded. It then waits until new call to `next()` function is issued. We will demonstrate this in the next section.

Generators and Iteration Protocol

PYTHON ACAD… / 15. [HOME] GENERATORS, REGEX & PACKAGE I… / GENERAT… / GENERATORS AND ITERATION PR…

In this section we will prove that the generator function does not produce all the values at once, as for example the `list()` function. Actually, **generator functions produce only generator objects**. Generator objects are those in fact, that are responsible for producing the values.

```
def my_range(start,stop,step=1):
    test = (lambda x,y: x<y and step>0, lambda x,y: x>y and step < 0)
[ start > stop]

    while test(start,stop):
        yield start
        start += step
```

We will demonstrate this by using `next()` function on generator object:

1. First we need to create a generator object. Calling a generator function creates a generator object. However, it does not start running the function. We store it in a variable `nums`:

```
>>> nums = my_range(2,10)
>>> nums
<generator object my_range at 0x7f617598d258>
```

2. Once we have the generator object, we can begin to generate values:

```
>>> next(nums)
2
>>> next(nums)
3
>>> next(nums)
4
```

3. We said, that generators follow the iteration protocol - therefore they can be used in for loops, for example. Iteration protocol dictates that once the iterator is exhausted (no more values to generate), `StopIteration` error should be raised

```
..
>>> next(nums)
8
>>> next(nums)
9
>>> next(nums)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Generators vs. Iterators

PYTHON ACADE… / 15. [HOME] GENERATORS, REGEX & PACKAGE IMPOR… / GENERATO… / GENERATORS VS. ITERAT…

Function, that returns a iterator object is `map()`. The returned object does not support use of function `len()`, but supports functions that make use of iteration protocol, e.g. `max()`. By calling `max()` on iterator object, the iterator is exhausted and any `next()` call will just return `StopIteration` error:

```
>>> m = map(lambda x: x, range(4))
>>> len(m)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'map' has no len()
>>> max(m)
3
>>> next(m)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

And the same is valid for generator objects:

No support for `len()` :

```
>>> nums = my_range(2,10)
>>> len(nums)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'generator' has no len()
```

But tools making use of iteration protocol work fine:

```
>>> 3 in nums
True
>>> max(nums)
9
>>> next(nums)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

How many yield statements?

PYTHON ACAD… / 15. [HOME] GENERATORS, REGEX & PACKAGE IMP… / GENERAT… / HOW MANY YIELD STATEME…

We are not limited to the use of only one yield statement. We can use as many as we want in our generator functions. In the example below, we yield paralelly values from two sequences:

```
def yield_from_two(seq1, seq2):
    while seq1 and seq2:
        print('Sending item from seq1:')
        yield seq1.pop()

        print('Sending item from seq2:')
        yield seq2.pop()
```

We are now able to extract data from two lists index-wise. That means, we get the first item from the `seq1` and after that we get the first item from the `seq2` and then the loop repeats until we have exhausted the first or the other list:

```
>>> g = yield_from_two(list(range(0,5)), list(range(5,10)))
>>> next(g)
Sending item from seq1:
4
>>> next(g)
Sending item from seq2:
9
>>> next(g)
Sending item from seq1:
3
>>> next(g)
Sending item from seq2:
8
```

We may notice that the code execution stops after any yield statement returns a given value.

Generator Return Statement

PYTHON ACAD… / 15. [HOME] GENERATORS, REGEX & PACKAGE IMP… / GENERAT… / GENERATOR RETURN STATE…

Until now, you may have gain an impression, that generator functions cannot contain `return` statement, because they already have `yield` statement. But that is not true. We can put a `return` statement wherever we want inside our generator functions. After all, function, that does not contain a `return` statement returns value `None`. And that is valid for generator functions as well. We will demonstrate this on the example from the previous section, into which we have added a `return` statement at the end:

```
def yield_from_two(seq1, seq2):
    while seq1 and seq2:
        print('Sending item from seq1:')
        yield seq1.pop()

        print('Sending item from seq2:')
        yield seq2.pop()

    return 'Lists have been exhausted'
```

And if we iterate over the generator object, we get a `StopeIteration` exception, which will containt the string, we have returned from our generator. That is actually the place, where the `return` statement sends the returned value:

```
>>> next(g)
Sending item from seq1:
3
>>> next(g)
Sending item from seq2:
1
>>> next(g)
Sending item from seq1:
4
>>> next(g)
Sending item from seq2:
2
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration: Lists have been exhausted
```

When Generators are Useful

Similarly as we have found use for `map` or `filter` objects, we can find use for generators. Compared to `map()` or `filter()`, generator functions allow us to pack much more code or better said, logic inside them.

1. Generators are used to **scan or generate very long collections of data**, e.g. huge text files.
2. We can as well create **inifinite value generators** that, compared to infinite while loops, generate their values, only when we ask them to do that.

For example we need a program to generate all permuations from unique 10 letters. That is 3628800 possibilities. We probably do not want to generate a list of all of them into computer's, but we rather want to explore each possibility alone.

3. Finally, due to the support of iteration protocol, generators allow us to **create pipelines** similar to those we created using `map()` and `filter()` functions.

For example, we could solve the task on email domain extraction using a chain of `map()` and `filter()` calls - the values are not actually generated at once, but they are pulled from the filter, through the two following maps up to the list function.

```
1  r = list(map(lambda x: x.rstrip('.,/\\; '), map(lambda x:
   x.split('@')[-1], filter(lambda x: '@' in x,text.split()))))
```

Infinite Generators

Infinite generator are usually created using infinite **while loop**. Let's say we would like to generate a sequence of items, but we would like to start from the beginning once we have arrived at the end of a given sequence.

Here is a simple example:

```
def cycle(seq):
    while seq:
        for item in seq:
            yield item
```

We could generate items infinitely long:

```
>>> g = cycle([1,2,3])
>>> next(g)
1
>>> next(g)
2
>>> next(g)
3
>>> next(g)
1
>>> next(g)
2
>>> next(g)
3
```

The reason to use a generator and not a normal function is that we do not want the values to be generated at once, but we rather prefer to take them out of the generator, when we need them. A good analogy is a card deck:

```
>>> deck = [type+str(value) for type in
['Clubs','Diamonds','Hearts','Spades'] for value in
['Ace']+list(range(2,11))+['Jack','Queen','King']]
>>> import random
>>> random.shuffle(deck)
```

Now we can serve cards, when the player decides:

```
>>> def serve(deck):
...     while True:
...         for card in deck:
...             yield card
...
```

And during the game, the player says: "Next please":

```
>>> c = serve(deck)
>>> next(c)
'Diamonds7'
>>> next(c)
'Hearts8'
>>> next(c)
'Hearts2'
```

Finally, there could be two programs that communicate over one file. One writes into the file and the other reads. Each of the programs does the input / output operation on its own pace. The result could be simulated as follows:

- writer - writes sometimes slower

```
def writer(file):
    file.seek(0,2)
    i = 1
    while True:
        file.write('Line number {}\n'.format(i))
        file.flush()
        i+=1
        time.sleep(randint(1,3))
```

- reader - checks for updated more often. If there are no new, it writes `'NO LINE'`

```
def reader(file):
    file.seek(0,2)
    while True:
        line = file.readline()
        if not line:
            print('NO LINE')
            time.sleep(1)
            continue
        else:
            yield line
```

Now we can open two terminal windows with the same current working directory. We paste the `writer()` function definition into terminal 1 and `reader()` definition into terminal 2. In each of the windows we should open the same file - one in `'w'` mode, the other in `'r'` mode:

- terminal 1

```
>>> f1 = open('test.txt', 'w')
```

- terminal 2

```
>>> f2 = open('test.txt')
```

Now we will run the function function `writer()` in the terminal 1 and iterate over generator returned by function `reader()` in the terminal 2:

- terminal 1 - the function will run infinitely so unless we press Ctrl+c, there will be one blank line following the function call

```
>>> writer(f1)
```

- terminal 2

```
>>> for line in reader(f2):
...     print(line)
...
```

You should now observe the communication between the two terminal windows across one file in the terminal 2.

- terminal 2

```
NO LINE
NO LINE
Line number 4
NO LINE
NO LINE
NO LINE
Line number 5
```

If you now stop the `writer()` function, the `reader()` will go on reading and printing `'NO LINE'`.

Here we can see two independent programs, each running at its own pace. We used `reader()` function to simulate a human that asks for another input for example. This request can come in different time points, but the important thing is, that the information is not read all at once, but dosed on request.

Generator Pipelines

PYTHON ACADE... / 15. [HOME] GENERATORS, REGEX & PACKAGE IMPORTI... / GENERATO... / GENERATOR PIPELIN...

Pipeline can be understood as a chain of function calls where output of one call is passed as a input into another call. This principle is often applied when using for example Bash commands in Linux terminal. We have seen this principle often when working with `map()` and `filter()` functions.

Now we will demonstrate it with help of generators. Let's say, we have many huge file we need to go over and extract a given information from them. However, their size is so huge, that we cannot use classic functions, that would try to return the whole content at once.

We need to set up a pipeline, where

1. the first component will find all the relevant filenames
2. the following member of the chain will open and generate the files
3. another component will read a one line from a file and will send it to another component,
4. the last component will do its job by filtering the lines of interest

Why cannot we do this in one function? Well we could, but the advantge of the above design is in component reusability. Each of those components can be used in another scenario, where the input object supports iteration protocol.

```
1 import os
2
3 def find_names(target_string, dir_path):
4     for dirpath, dirnames, filenames in os.walk(dir_path):
```

```
        for filename in filenames:
            if target_string in filename:
                yield os.path.join(dirpath,filename)

def open_files(names):
    for name in names:
        yield open(name)

def read_file(files):
    for file in files:
        for line in file:
            yield line

def extract_lines(target,lines):
    for line in lines:
        if target in line:
            yield line
```

Now, creating the pipeline, we can go extracting line by line from a given directory structure. Below we extract all the lines that contain word "python" in them:

```
>>> lines =
extract_lines('python',read_file(open_files(find_names('.txt','/home/martin
```

Result:

```
>>> next(lines)
'python tutor\n'
>>> next(lines)
'\t\t\t- Run python console\n'
>>> next(lines)
'\t\t\t- Experiment with python console\n'
```

Generator Expressions

Besides writing generator functions, that contain `yield` keyword, there is another way, how generator objects can be created. Sometimes, we may want to generate sequence as we did with comprehensions, but we cannot afford to build the whole sequence at once. That is the moment when **generator expressions** come to play.

Generator expressions follow the same syntax rules as list comprehensions. The only difference is that we enclose the whole expression in parentheses and not square brackets:

```
>>> text = 'Lorem ipsum dolor sit amet, Consectetur adipiscing elit, sed
do eiusmod Tempor incididunt ut Labore et dolore magna aliqua.'
>>> title_cased = (word for word in text.split() if word.istitle())
>>> next(title_cased)
'Lorem'
>>> next(title_cased)
'Consectetur'
>>> next(title_cased)
'Tempor'
```

The generator expression should not be filled with sophisticated statements. Rather it is a lightweight equivalent to generator function, which we should use if we need to include more complex logic inside our generator object, that produces values on demand.

Intro

Regular expressions (regex) are strings using agreed characters to express various **patterns**. These patterns are then searched for. An example could be a search for all emails in a given input string. Individual emails have to be unique, but at the same time, they have to follow a given pattern. And this is the pattern we can define, using regular expressions.

Characters used as special symbols in regex are special only if interpreted by dedicated functions defined in Python **re** module.

The example below looks for the first word, that start with letter `'t'` :

```
>>> import re
>>> text = 'The tropical thunder Irma'
>>> pattern = 't\w+'
>>> match = re.search(pattern, text)
>>> match
<_sre.SRE_Match object; span=(4, 12), match='tropical'>
```

The special characters in the `pattern` variable were `\w` and `+` . In the following few sections, we will explain their meaning and the meaning of other special regex characters.

Basic Search - Single Match

To search with help of regular expressions, we need to have the following ingredients:

1. imported module `re`
2. string we want to search through
3. pattern we want to search
4. call the searching function from the `re` module

The most basic pattern we can imagine using as regular expression is the actual sequence of characters we want to find in a given string. We will learn to use the first function from `re` module - `search()` .

Function `re.search()` returns so called `Match` object, if the given pattern has been found, otherwise value `None` is returned. This function takes at least two arguments:

1. regex pattern,
2. string we want to search through.

Let's say we would like to find string `'day'` in the string `'It is raining already for two days now'`

```
>>> import re
>>> text = 'It is raining already for two days now'
>>> pattern = 'day'
>>> result = re.search(pattern,text)
```

If we printed the content of the variable `result` , we get quite few information about the:

1. span of the matching string - start and end indices
2. the matched string

```
>>> result
<_sre.SRE_Match object; span=(30, 33), match='day'>
>>> result.start()
30
>>> result.end()
33
```

Now we can extract the matched string:

```
>>> text[result.start():result.end()]
'day'
```

Basic Search - Multiple Match

In the previous section, using `search()` function, we got only the first string, that matched a given pattern of characters.

To demonstrate the multiple match, we will again search for the patterns `'day'`, this time in a string: `'It is raining already for two days now, but yesterday, the weather was much worse.'`

```
>>> import re
>>> text = 'It is raining already for two days now, but yesterday, the
weather was much worse.'
>>> pattern = 'day'
```

To get all the strings, that match the given pattern, we can use:

1. `findall()` - returns a list of matching strings

```
>>> result = re.findall(pattern,text)
>>> result
['day', 'day']
```

2. `finditer()` - returns a iterator object, that generates `Match` objects (not strings) on request. Compared to `findall()`, `finditer()` gives us more flexibility to:
    - Work with huge strings and despite the big number of matches, not to generate them all at once into a list
    - Generate `Match` objects, that provide much more functionality as we will learn soon

```
>>> result = re.finditer(pattern,text)
>>> result
<callable_iterator object at 0x7fea31907160>
```

```
>>> list(result)
[<_sre.SRE_Match object; span=(30, 33), match='day'>, <_sre.SRE_Match
object; span=(50, 53), match='day'>]
```

Compiling Regex Pattern

We have omitted one of additional step in our regex workflow - regex compilation. Regex compilation takes place using `re.compile()` function. The function takes as input the regex string and returns compiled regex `Pattern` object.

The `search()`, `findall()`, `finditer()` and other functions, that accept the regex string, have to perform the regex string compilation as well. With each call a new compiled representation is stored. And so, if the same regex again and again is used, we begin to store unnecessary compiled copies of the same pattern. It is therefore more efficient to compile the expressions a program uses frequently and perform the search using these.

We will therefore follow this recommendation in the following examples:

```
>>> regex = re.compile('day')
>>> regex
re.compile('day')
>>> type(regex)
<class '_sre.SRE_Pattern'>
```

And now we can use `Pattern` object's own `search()`, `findall()`, `finditer()` methods:

```
>>> text
'It is raining already for two days now, but yesterday, the weather was
much worse.'
```

- `search()`

```
>>> regex.search(text)
<_sre.SRE_Match object; span=(30, 33), match='day'>
```

- `findall()`

```
>>> regex.findall(text)
['day', 'day']
```

- `finditer()`

```
>>> regex.finditer(text)
<callable_iterator object at 0x7fea319071d0>
```

Special Regex Characters

PYTHON ACAD… / 15. [HOME] GENERATORS, REGEX & PACKAGE I… / REGULAR EXPRESS… / SPECIAL REGEX CHARA…

So far, the pattern we have used was not special at all. We search for a sequence of three chacters `'day'` in a string. We could have done the same work with `str.find()` method. The real regex power comes with use of special characters. We will guide you through the following sets of special characters - characters that determine:

1. the number of times a a subpattern may repeat in the target string (`*` , `+` , `?` , `{2,3}`)
2. what characters may appear at a given position in the target string (`[A-Z]` , `[a-z]` , `[,./]` ,...)
3. what character types (digits, letters, spaces etc.) may appear at a given position in the target string (`\d` , `\w` , `\s` , `.`)
4. where given pattern may appear - at the beginning or end of the searched string or the beginning / end of a word (`^` , `$` , `\b`)

... and more

Character Repetitions

PYTHON ACAD… / 15. [HOME] GENERATORS, REGEX & PACKAGE IM… / REGULAR EXPRESSI… / CHARACTER REPETIT…

The first group of special regex symbols to be discussed are those, that determine repetition. We will demonstrate the their workings on the string `aabbbaaabaabba`.

The flags to be demonstrated are:

FLAG	MEANING
`+`	Sequence occurs at least once
`*`	Sequence occurs zero or more times (that is, it is optional)
`?`	Sequence occurs zero times or once
`{m,n}`	Sequence occurs from to `m` to `n` times

The flag has to follow the sequence its repetiton it defines.

Task for `+`: extract all sequences that begin with letter `a` and followed by one or more letters `b`

```
>>> text = 'aabbbaaabaabba'
>>> regex = re.compile('ab+')
>>> regex.findall(text)
['abbb', 'ab', 'abb']
```

Task for `*`: extract all sequences that begin with letter `a` and followed by zero or more letters `b`

```
>>> text = 'aabbbaaabaabba'
>>> regex = re.compile('ab*')
>>> regex.findall(text)
['a', 'abbb', 'a', 'a', 'ab', 'a', 'abb', 'a']
```

Task for `?`: extract all sequences that begin with letter `a` and followed by zero or one letter `b`

```
>>> text = 'aabbbaaabaabba'
>>> regex = re.compile('ab?')
>>> regex.findall(text)
['a', 'ab', 'a', 'a', 'ab', 'a', 'ab', 'a']
```

Task for `{m,n}` : extract all sequences that begin with letter `a` and is followed by 2 to 3 letters `b`

```
>>> text = 'aabbbaaabaabba'
>>> regex = re.compile('ab{2,3}')
>>> regex.findall(text)
['abbb', 'abb']
```

Once again we have to stress the fact, that the repetition flag specifies the repetition only for the preceding character - letter `'b'` .

Character Sets

PYTHON ACADE… / 15. [HOME] GENERATORS, REGEX & PACKAGE IMPOR… / REGULAR EXPRESSIO… / CHARACTER SE…

Character that could match at a given point of the target string are enclosed inside square brackets `[ ]` and are called character sets.

Let's say we would like to extract all the sequences that begin with `'t'` or `'w'` followed by letter `'e'` in the string:

`'It is raining already for two days now, but yesterday, the weather was much worse.'`

That is, we are looking for the following pattern:

```
>>> text = 'It is raining already for two days now, but yesterday, the
weather was much worse.'
>>> pattern = '[tw]e'
>>> regex = re.compile(pattern)
```

```
>>> regex.findall(text)
['te', 'we']
```

If there are more characters we want to allow to be matched in a given place, we can use a **character range** including the `-` sign:

- `[A-Z]` - match any of the uppercase letters 'A' to 'Z'
- `[a-z]` - match any of the lowercase letters 'a' to 'z'
- `[0-9]` - match any of the digits between '0' to '9'

Now we can make our regex pattern more useful and search for any word that begins with `'t'` or `'w'` in the target string:

```
>>> pattern = '[tw][a-z]+'
>>> regex = re.compile(pattern)
```

```
>>> regex.findall(text)
['two', 'terday', 'the', 'weather', 'was', 'worse']
```

Well, some of the sequences identified are not valid words yet, but we will learn how to extract only words in the following section.

We still need to speak about the case, when not all characters from a range are acceptable for our pattern. In that case we need to use the symbol `^` inside the square brackets to tell, which character should not be matched. The `^` symbol should be the first one in the character set.

This time we will search throught the string:

```
'This is my day-to-day routine'
```

We would like to extract all the words, excluding the `-` sign:

```
>>> pattern = '[A-Za-z]+[^ -]*[A-Za-z]+'
>>> regex = re.compile(pattern)
```

```
>>> regex.findall(text)
['This', 'is', 'my', 'day', 'to', 'day', 'routine']
```

The pattern `'[A-Za-z]+[^ -]*[A-Za-z]+'` can be translated as:

"Give me all the sequences that begin with one or more upper or lowercase letters, then there should not be any blank spaces or minus signs and then the sequence should continue with one or more upper or lowercase letters."

In the middle of our pattern we have included also blank space, in order we extract the words separately. Excluding only minus sign would have the following result:

```
>>> re.findall('[A-Za-z]+[^-]*[A-Za-z]+',text)
['This is my day', 'to', 'day routine']
```

And omitting the `*` repetition symbol would mean, that the extracted words should have at least three characters - there are three character sets in our pattern.

```
>>> re.findall('[A-Za-z]+[^ -][A-Za-z]+',text)
['This', 'day', 'day', 'routine']
```

Character Types

Besides the character sets and their ranges, regular expressions define special characters that will match a given number of:

1. Digits (`\d`) and non-digits (`\D`),
2. alphanumeric (`\w`) and non-alphanumeric (`\W`) characters,
3. white space characters - newline, tab, space etc. (`\s`) and non-whitespace character (`\S`),
4. any character except a newline (`.`) - dot

And this is how, they are used:

Digits \d

Task: Extract all the sequences that follow the correct time formatting pattern HH:MM (e.g.: 13:05)

We want to create a pattern that will say:

"String that begins with exactly two digits, followed by colon `:` and terminated with two other digits"

```
text = ' Yesterday, at 13:05, sun was shining. At 17:05 it began to
rain.'
pattern = '\d{2}:\d{2}'
```

```
regex = re.compile(pattern)
```

```
>>> regex.findall(text)
['13:05', '17:05']
```

Non-digits \D

Task: Extract all the sequences that do not contain numbers.

We have a sequence taken from an URL request:

```
text
="TIME%10IS%10RUNNING%10OUT%10TO%10CLAIM%10THIS%10FREE%10EBOOK%1011%3"
regex = re.compile('\D+')
```

And we will be able to make the string more readable:

```
>>> regex.findall(text)
['TIME%', 'IS%', 'RUNNING%', 'OUT%', 'TO%', 'CLAIM%', 'THIS%', 'FREE%',
'EBOOK%', '%']
```

Alpanumeric chars \w

We will often use special character, that matches all the alphanumeric characters, when we want to **get rid of whitespaces or punctuation**:

```
text = ' Yesterday, at 13:05, sun was shining. At 17:05 it began to
rain.'
regex = re.compile('\w+')
```

```
>>> regex.findall(text)
['Yesterday', 'at', '13', '05', 'sun', 'was', 'shining', 'At', '17',
'05', 'it', 'begin', 'to', 'rain']
```

Be careful, we are still not able to extract individual words, rather sequences of given character type. Therefore the pattern `\w{5}` will not extract words containing 5 characters:

```
>>> regex = re.compile('\w{5}')
>>> regex.findall(text)
```

```
['Yeste', 'shini', 'begin']
```

Non-alphanumeric chars \W

If we by change want to collect sequences that will contain non-alphanumeric character, we use `\W` :

```
text = ' Yesterday, at 13:05, sun was shining. At 17:05 it began to
rain.'
regex = re.compile('\w\W+\w')
```

```
>>> regex.findall(text)
['y, a', 't 1', '3:0', '5, s', 'n w', 's s', 'g. A', 't 1', '7:0', '5 i',
't b', 'n t', 'o r']
```

Our pattern above says:

"Extract all the sequences, that contain three characters and begin and end with alpanumeric character. In the middle of the sequence there should be **at least** one non-alphanumeric character."

Whitespace chars \s

```
text = ' Yesterday, at 13:05, sun was shining. At 17:05 it began to
rain.'
regex = re.compile('\w+\s')
```

We extracted all the sequences that contain one or more alphanumeric characters folowed by a whitespace character.

```
>>> regex.findall(text)
['at ', 'sun ', 'was ', 'At ', '05 ', 'it ', 'begin ', 'to ']
```

Non-whitespace chars \S

```
text = ' Yesterday, at 13:05, sun was shining. At 17:05 it began to
rain.'
regex = re.compile('\w+\S')
```

```
>>> regex.findall(text)
['Yesterday,', 'at', '13:', '05,', 'sun', 'was', 'shining.', 'At', '17:',
'05', 'it', 'begin', 'to', 'rain.']
```

Any character except newline '.'

If our searched string contained a newline character `\n`, the match would stop at the position of this character. In the example below we get two matches of sequences, that contain at least one character that is not newline.

```
text = 'Yesterday, at 13:05, sun was shining.\nAt 17:05 it began to
rain.'
regex = re.compile('.+')
```

```
>>> regex.findall(text)
['Yesterday, at 13:05, sun was shining.', 'At 17:05 it began to rain.']
```

Anchor Symbols

PYTHON ACADE… / 15. [HOME] GENERATORS, REGEX & PACKAGE IMPOR… / REGULAR EXPRESSI… / ANCHOR SYMB…

In this section we will finally learn, how to extract individual words from a sentence. To do that, we need to specify, what type of the character should the target pattern begin or end with.

We are able to do that using anchor characters:

- `\b` - marks the **beginning or end** of a word (sequence of alphanumeric characters)
- `\B` - marks **not the beginning or end** of a word
- `^` - marks the **begining** of a string or line (lines are terminated with `\n`)
- `\A` - marks the **beginning** of a string
- `$` - mars the **end** of a string or a line
- `\Z` - marks the **end** of a string

And here are the examples:

- `\b`

If we want to use special character `\b`, we need to enclose it in a **raw string** or we need to escape it with one more backslash - `\\b`.

All the examples below extract all the words in the string:

```
'Yesterday, at 13:05, sun was shining.\nAt 17:05 it began to rain.'
```

Task: Extract all the sequences that begin with one or more alphanumeric characters:

```
>>> regex = re.compile(r'\b\w+')
>>> regex.findall(text)
['Yesterday', 'at', '13', '05', 'sun', 'was', 'shining', 'At', '17',
'05', 'it', 'began', 'to', 'rain']
```

Task: Extract all the sequences that end with one or more alphanumeric characters

```
>>> regex = re.compile(r'\w+\b')
>>> regex.findall(text)
['Yesterday', 'at', '13', '05', 'sun', 'was', 'shining', 'At', '17',
'05', 'it', 'began', 'to', 'rain']
```

Below the example using two backslashes:

```
>>> regex = re.compile('\\b\w+')
>>> regex.findall(text)
['Yesterday', 'at', '13', '05', 'sun', 'was', 'shining', 'At', '17',
'05', 'it', 'began', 'to', 'rain']
```

- `\B`

Task: Extract all the sequences that contain one or more alphanumeric characters but do not end with character `'i'`

```
>>> regex = re.compile(r'\w+i\B')
>>> regex.findall(text)
['shini', 'rai']
```

- `^`, `\A`

At the moment, these two characters will have the same impact on the result - they will mark the beginning of the searched string

Task: Extract the sequence at the beginning of the searched string, if it beginns with letters `'Y'` or `'A'` .

```
>>> regex = re.compile(r'^[YA]\w+')
>>> regex.findall(text)
['Yesterday']
>>> regex = re.compile(r'\A[YA]\w+')
>>> regex.findall(text)
['Yesterday']
```

- `$` or `\Z`

Again these two symbols will for now behave the same marking the end of the searched string.

Task: Extract the last word of the sentence

```
>>> regex = re.compile('\w+\S*\Z')
>>> regex.findall(text)
['rain.']
>>> regex = re.compile('\w+\S*$')
>>> regex.findall(text)
['rain.']
```

Special Characters Inside Char Sets

PYTHON ACA... / 15. [HOME] GENERATORS, REGEX & PACKA... / REGULAR EXPRE... / SPECIAL CHARACTERS INSIDE ...

It is important to note, that once the special characters `.` , `$` are placed inside the square brackets, they loose their special magic and regex compiler interprets its characters literally:

```
>>> text ='Yesterday, at 13:05, sun was shining.\nAt 17:05$$ it began to
rain.'
>>> regex = re.compile('[$]+')
>>> regex.findall(text)
```

```
['$$']
>>> regex = re.compile('[.]+')
>>> regex.findall(text)
['.', '.']
```

The character `^` has different meaning inside and outside square brackets.

Below we extract sequences containing 5 characters excluding b, c and d.

```
>>> regex = re.compile('[^bcd]{5}')
>>> regex.findall(text)
['Yeste', 'ay, a', 't 13:', '05, s', 'un wa', 's shi', 'ning.', '\nAt 1',
'7:05$', '$ it ', 'egan ', 'to ra']
```

In the below example we are trying to match the beginning of the matched string with one of the letters b,c or d followed by exactly 5 alphanumeric characters.

```
>>> regex = re.compile('^[bcd]\w{5}')
>>> regex.findall(text)
[]
```

On the other hand, characters like `\w`, `\d`, `\s` keep their powers:

```
>>> regex = re.compile('[\w]+')
>>> regex.findall(text)
['Yesterday', 'at', '13', '05', 'sun', 'was', 'shining', 'At', '17',
'05', 'it', 'began', 'to', 'rain']
```

```
>>> regex = re.compile('[\d]+')
>>> regex.findall(text)
['13', '05', '17', '05']
```

```
>>> regex = re.compile('[\s]+')
>>> regex.findall(text)
[' ', ' ', ' ', ' ', ' ', '\n', ' ', ' ', ' ', ' ', ' ']
```

Escaping Special Characters

It is good to know, that if we wanted to match a sequence of characters that could contain special regex characters, we need to escape those in the regex pattern:

```
>>> text = 'Special chars are \w, \d, \s'
```

If printed, we notice, that the original string now contains escape backslashes:

```
>>> text
'Special chars are \\w, \\d, \\s'
```

Therefore our regex pattern has to contain two backslashes as well:

```
>>> regex = re.compile(r'\\.')
>>> regex.findall(text)
['\\w', '\\d', '\\s']
```

Otherwise there would be no match:

```
>>> regex = re.compile(r'\.')
>>> regex.findall(text)
[]
```

Matching Algorithm is Greedy

We may have already noticed that when we ask regex pattern to match a sequence, it matches as much as possible until it stops.

To demonstrate how many characters does the greedy vs. non-greedy match eat until the match is stopped, we will use the following function:

```
1 def visualize_matches(pattern, text):
2     print(text + '\n')
3
```

```
    for match in re.finditer(pattern, text):

        start,end = match.start(), match.end()

        print('.'*start + text[start:end])
```

This function will show us the original searched string and position of each match found. The position is depicted as offset using dots, for each preceding character in the original string.

We will search the string:

```
text = 'aabbbabbaaabba'
```

One or more '+'

For example, in the demonstration below, 3 matches were found - at indices 1, 5 and 10:

```
>>> visualize_matches('ab+',text)
aabbbabbaaabba
.abbb
.....abb
..........abb
```

The above search matches all the letters `'b'` until it encounters next letter `'a'` . The algorithm is matching the upper bound of its possibilities, given by the `+` character - one or more characters `'b'` .

The non-greedy equivalent of `ab+` is that when the lower bound of the condition is met - one character `'b'` . To tell Python, we want the non-greedy version, we have to append the `?` sign following the `+` sign:

```
>>> visualize_matches('ab+?',text)
aabbbabbaaabba
.ab
.....ab
..........ab
```

Zero or more '*'

Greedy variant matches all it can - single `'a'` as well as many `'a'` followed by `'b'` as possible.

```
>>> visualize_matches('ab*',text)
aabbbabbaaabba
a
.abbb
.....abb
........a
.........a
..........abb
..............a
```

The non-greedy version will match the lower bound - zero of `'b'` and then it is done. Therefore only charcters `'a'` are matched.

```
>>> visualize_matches('ab*?',text)
aabbbabbaaabba
a
.a
.....a
........a
.........a
..........a
..............a
```

Zero or one '?'

Here similarly, the lower bound of the condition, which is zero, will be followed in the non-greedy version:

```
>>> visualize_matches('ab?',text)
aabbbabbaaabba
a
.ab
.....ab
........a
.........a
```

```
..........ab
............a
```

```
>>> visualize_matches('ab??',text)
aabbbabbaaabba
a
.a
.....a
........a
........a
..........a
............a
```

Match Groups

Regular expressions provide us with capability of isolating substrings inside the matched string through the mechanism of so called groups. Groups are signaled in the regex pattern by being enclosed inside parentheses `()` .

Let's say, we search for an an email address that consists of two parts (`local-part@domain`). At the same time, we would like to isolate local part and the domain.

We will use a very simple pattern to identify the email and in no means is this a regular expression that would match any possible email out there.

```
>>> text = "My email is support_admin@company.com, but I do not use it
often"
>>> regex = re.compile('([\w_.-]+)@([\w-]+)\.(\w+)')
>>> r = regex.search(text)
>>> r
<_sre.SRE_Match object; span=(12, 37), match='support_admin@company.com'>
```

Our regular expression contains three groups (local part, domain separated by dot `\.` into subdomain, e.g. "company", top-level domain, e.g. "com"):

```
([\w_.-]+)  @  (\w+)  \.   (\w+)'
|________|     |____|      |____|
  local p.   sub-domain   top-level domain
```

To access the individual parts of the matched string, we can use `groups()` method, that will return a tuple of substrings:

```
>>> r.groups()
('support_admin', 'company', 'com')
```

Or we can access each group by its index - the first group at index 1:

```
>>> r.group(1)
'support_admin'
>>> r.group(2)
'company'
>>> r.group(3)
'com'
```

The group at the zeroth index will is the whole matched string:

```
>>> r.group(0)
'support_admin@company.com'
```

Alternative Patterns

PYTHON ACAD… / 15. [HOME] GENERATORS, REGEX & PACKAGE IMP… / REGULAR EXPRESSI… / ALTERNATIVE PATTE…

We can use the pipe symbol `|` to separate two patterns and indicate that either pattern should match. The pipe symbol is often used in connection to groups

When one pattern completely matches, that branch is accepted. This means that once A matches, B will not be tested further, even if it would produce a longer overall match.

```
>>> text
'aabbbabbaabba'
```

We got three matches for the following:

```
>>> regex = re.compile('a(a+|b+)')
>>> visualize_matches(regex,text)
aabbbabbaabba
aa
.....abb
........aa
```

Each match has to begin with letter `a` and will contain one group (there is only one pair of parentheses in the regex pattern):

```
>>> r[0].groups()
('a',)
>>> r[1].groups()
('bb',)
>>> r[2].groups()
('a',)
```

We can see in the second match's (`abb`) case, that as there was no available letter a at index 6, the matching algorithm took the second branch, matching all the `b` s possible until it hits another letter `a` .

What is a Package?

Package is a directory that contains python files. In more complex projects, python files are grouped inside directories based on their functionality they cover. In this case python files are in separate directories which can be nested inside other directories.

The possibility to create packages is, when we need to isolate different groups of python files. This need arises, for example, when building a web application or when building a more complicated set of Python modules.

Example of python package (directory tree):

```
top_dir
......main.py
......level1_dir1
............\__init\__.py
............math_tools.py
............string_tools.py
............level2_dir1
................\__init\__.py
................search_tools.py
......level1_dir2
............\__init\__.py
............other_tools.py
```

What Makes a Package?

The best known Python package to us is currently `requests`. If we looked into `site-packages` folder in our Python installation, we would find a directory called `requests`. Inside this directory, there are multiple Python files:

```
$ tree requests
requests
├── adapters.py
├── api.py
├── auth.py
├── certs.py
├── compat.py
├── cookies.py
├── exceptions.py
├── help.py
├── hooks.py
├── __init__.py
├── _internal_utils.py
├── models.py
├── packages.py
├── __pycache__
├── sessions.py
├── status_codes.py
├── structures.py
├── utils.py
└── __version__.py
```

One of them is of particular interest to us at the moment - the `__init__.py` file. This is the file that allows us to write `import requests` in our Python code. If you take a good look, inside the `requests` folder, there is no `requests.py` file. Now we know that `requests` is a package, but the question is, what actually happens, when we tell Python to import a package.

Importing a Package

We will now continue with the example of importing `requests` package. We have written command `import requests` and Python executes it as follows:

1. Python performs classic module search (`sys.modules` -> `sys.path`)
2. The directory named `requests` is found in the `site-packages` folder
3. Python gives attention to this folder because it has the name of a module we want to import as well as it contains a `__init__.py` file
4. **Python loads - executes the `__init__.py` file**

So what is actually imported here is not the `requests.py` file (it does not exist), but the `__init__.py` file. So in order to understand what is actually being imported, we need to look inside `__init__.py` .

Inside the __init__.py

So we have opened the `requests/__init__.py` file and now we are trying to get our head around its content. When looking at such file, we may want to prefer to follow some strategy. We may want to answer the following questions:

1. Are there any functions defined inside the file? What do they do?
2. What imports are performed inside the file?
3. Are there any variables begining with single underscore or `__all__` variable?

In my `requests/__init__.py` file, there is one function defined, called `check_compatibility()` . This function checks the compatibility of the version of another Python package - `urllib3` . Actually, `requests` package makes heavy use of `urllib3` . There are also other module imports inside the `__init__.py` file, which are then used in its code.

Then, the imports. We can imagine the `__init__.py` file as a kind of a hub, that loads a lot of functionality from different Python modules, in order we do not have to import them all in our scripts. We just write `import requests` and everything, that is imported into the `__init__.py` file, is loaded with the package import into our program.

So if we open Python console in our terminal and import `requests` package:

```
>>> import requests
```

We have for example access to `RequestsDependencyWarning` class that is imported in the `__init__.py` file:

```
>>> requests.RequestsDependencyWarning
<class 'requests.exceptions.RequestsDependencyWarning'>
```

Or we have access to the package `urllib3` through the `requests` package:

```
>>> requests.urllib3
<module 'urllib3' from '/home/martin/anaconda3/lib/python3.5/site-
packages/urllib3/__init__.py'>
```

Package Import Forms

PYTHON ACAD… / 15. [HOME] GENERATORS, REGEX & PACKAGE IMP… / PACKAGE IMPORT… / PACKAGE IMPORT FO…

Sometimes, we may find import statements, that may look strange at the first sight. For example, when we want to import `HTTPBasicAuth` class from the `requests` package, we do it as follows:

```
1 from requests.auth import HTTPBasicAuth
```

We then can create a object, that will bear our username and password to the server, when sent with the request.

What is important is the form, we used to import this class. This class we will not find in the `requests/__init__.py` file, but in another file called `auth.py` inside the `requests` folder.

You can go and check it out, how does it look like.

So in case, we would like not to import the whole `requests` package but just:

1. single module inside the package:

```
# Syntax: import package.module
import requests.auth

# Syntax: from package import module
from requests import auth
```

2. single object / variable defined in a module in a package:

```
# Syntax: from package.module import attribute
from requests.auth import HTTPBasicAuth
```

If our package would be more complicated, meaning, it would contain more inner directories with `__init__.py` files, the import chain can be built as follows:

`import package0.package1.package2.module1` etc.

Package Relative Imports

PYTHON ACAD... / 15. [HOME] GENERATORS, REGEX & PACKAGE IM... / PACKAGE IMPOR... / PACKAGE RELATIVE IMP...

There is another way, how imports can be written in Python files inside packages. These are relative imports and can be recognized by dot or dots preceding the module name. The principle is actually an analogy to relative file paths.

In `requests` package, we may see the following import:

```
from .models import Request, Response, PreparedRequest
```

This import says - in the current package (folder), go to the modue `models` and import classes `Request`, `Response`, `PreparedRequest`.

The point of relation for a given relative import is the package in which the importing file resides.

The single dot tells us, that the module should be searched in the current folder (package). Two dots - `..module` - would say that the module should be searched in the parent directory of the current package. This situation does not occur with `requests` package as there is only one level of directories - the `requests` directory.

Generators

PYTHON ACADEMY / 15. [HOME] GENERATORS, REGEX & PACKAGE IMPORTING / QUIZ / GENERATORS

1/6

What are generators used for?

A. Create generator functions

B. Generate values on request

C. Create generator expressions

D. Generate all the values at once

Regular Expressions

1/10

What are regular expressions used for?

A. String formatting

B. Search for string patterns

C. Check, whether a given Python expression is correct

D. String encoding

Package Import

1/4

What do we need in order to make a directory a python package?

A. The directory has to contain file called __init__.py

B. The directory has to contain folder called __pycache__

C. The directory has to contain file called dir.py

D. The directory has to contain file called __import__.py

Game of Life

As a final task for this course, you can make your own implementation of so called Game of Life. To learn, what game of life is and how it can look like as a running program, visit the webpage of:

1. Stanford University page
2. or Wikipedia

Once you will learn, what are the rules of Game of Life simulation you can begin to implement:

1. The logic
2. Interface

Regarding the interface, you will want display the Game of Life directly in :

1. the terminal window - you may be interested in learning how package, called **colorama** works. Check its website at pypi
2. **pygame** based window - you can install a Python framework that is used for game development. This framework will help you to create your own window, in which the game will run. See the pygame website

Code Solution

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

Click to see our solution

DALŠÍ LEKCE