Week 11

Advanced Course in Programming

23.11.2023

Last week

Inheritance

About visibility

self

Operator overloading

A list out of a list

Let's assume the following problem:

We have a list where all items are strings. Each string contains digits only. We want to create a list of integers out of this.

Traditional solution

Create an empty list

Write a for statement for iterating all items one by one. Convert the items into integers with int function.

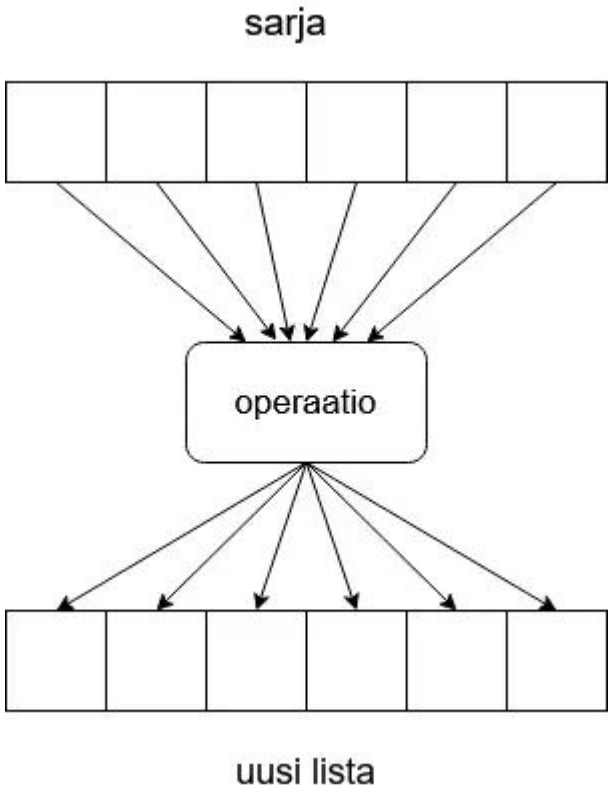
Add to new list with the append method

List comprehension

The idea is that we can create a new list with a single expression

Syntax

```
[<expression> for <item> in <sequence>]
```



For example

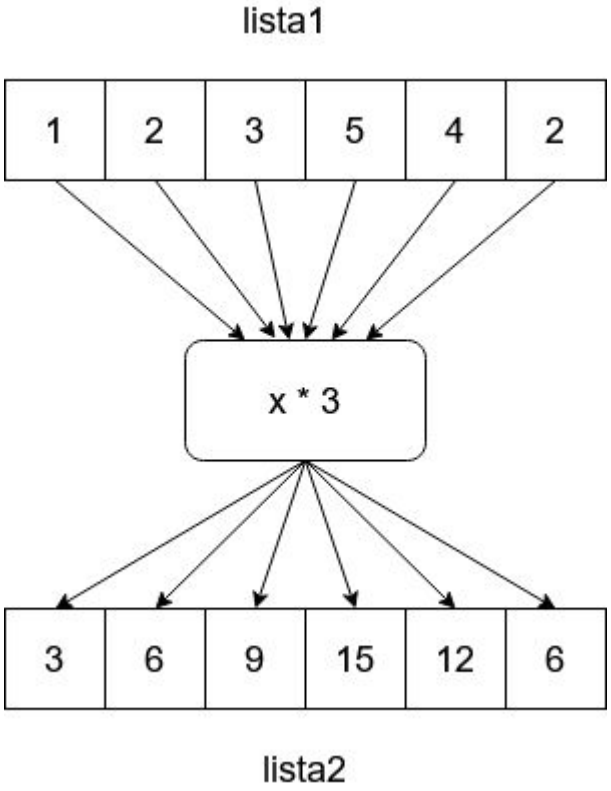
Same thing as a list comprehension:

```
[int(item) for item in number_list]
```

Example

```
lista1 = [1, 2, 3, 5, 4, 2]

lista2 = [x * 3 for x in lista1]
```



Calling own function in comprehension

```
def factorial(n: int):
    """ The function calculates the factorial n! for integers above zero """
    k = 1
    while n >= 2:
        k *= n
        n -= 1
    return k

if __name__ == "__main__":
    numbers = [5, 2, 4, 3, 0]
    factorials = []
    for number in numbers:
        factorials.append(factorial(number))
    print(factorials)
```

Filtering items

General syntax:

```
[<expression> for <item> in <series> if <Boolean expression>]
```

With function

```
def factorial(n: int):
    """ The function calculates the factorial n! for integers above zero """
    k = 1
    while n >= 2:
        k *= n
        n -= 1
    return k

if __name__ == "__main__":
    numbers = [-2, 3, -1, 4, -10, 5, 1]
    factorials = [factorial(number) for number in numbers if number >= 0]
    print(factorials)
```

Alternative branch

The if expression which we discussed earlier...

```
<expression1> if <condition> else <expression2>
```

Alternative branch

...may be combined with list comprehension:

```
[<expr.1> if <condition> else <expr.2> for item in sequence]
```

Comprehension and strings

The iterable sequence
can also be e.g. a string:

```
name = "Peter Python"

uppercased = [character.upper() for character in name]
print(uppercased)
```

Comprehension and strings with join method

Useful, when the join method is combined:

```
test_string = "Hello there, this is a test!"

vowels = [character for character in test_string if character in "aeiou"]
new_string = "".join(vowels)

print(new_string)
```

Comprehension, join and split

```
sentence = "Sheila keeps on selling seashells on the seashore"

sentence_no_initials = " ".join([word[1:] for word in sentence.split()])
print(sentence_no_initials)
```

Dictionary comprehension

```
sentence = "hello there"

char_counts = {character : sentence.count(character) for character in sentence}
print(char_counts)
```

Functions can call each other

```python
def hello(name : str):
    print("Hello", name)

def hello_many_times(name : str, times : int):
    for i in range(times):
        hello(name)
```

Functions can call each other

This, however, leads to infinite loop:

```
def hello(name : str):
    print("Hello", name)
    hello(name) # function calls itself
```

Recursion

Function calls itself with changing input

```
def fill_list(numbers: list):
    """ If the length of the list is less than 10, add items to the list """
    if len(numbers) < 10:
        numbers.append(0)
        # call the function again
        fill_list(numbers)
```

Recursion and return values

Immutable values should be returned from recursive functions:

```
def factorial(n: int):
    """ The function calculates the factorial n! for n >= 0 """
    if n < 2:
        # The factorial for 0 and 1 is 1
        return 1

    # Call the function again with an argument that is one smaller
    return n * factorial(n - 1)
```

Example: Fibonacci

```
def fibo(n: int):
    if n <= 2:
        return 1

    return fibo(n - 1) + fibo(n - 2)
```

If the function is called with 1 or 2 as its argument, it returns 1, as dictated by the condition n <= 2.

If the argument is 3 or greater, the function returns the value of fibonacci(n - 1) + fibonacci(n - 2). If the argument is 3 exactly, this value is equal to fibonacci(2) + fibonacci(1), and we already know the result of both of those from the previous step. 1 + 1 equals 2, which is indeed the third number in the Fibonacci sequence.

If the argument is 4, the return value is fibonacci(3) + fibonacci(2), which we now know to be 2 + 1, which equals 3.

If the argument is 5, the return value is fibonacci(4) + fibonacci(3), which we now know to be 3 + 2, which equals 5.

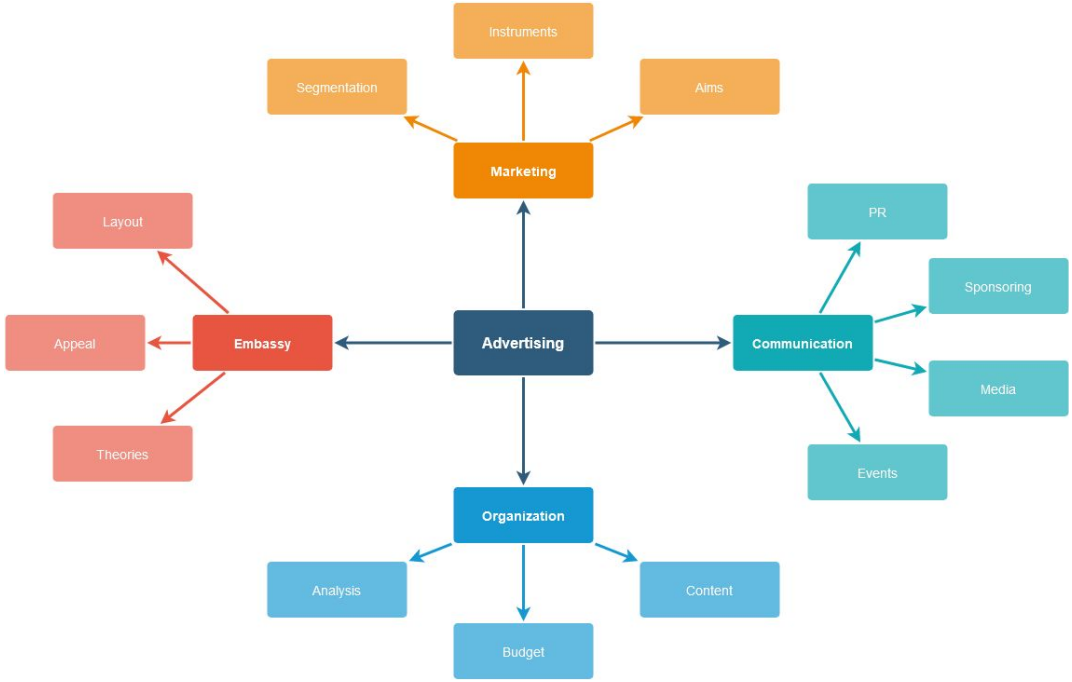
etc. etc.

Recursion: why and when?

Useful when dealing with structures which are recursive by nature

Trees, graphs, etc.

Often difficult to perceive at first



Next week

Functions as parameters

Lambda expressions

Generator functions

Functional programming: map, filter, reduce

Regular expressions