Week 12

Advanced Course in Programming

30.11.2023

Last Week

List comprehension

Filtering items in comprehensions

Dictionary comprehensions

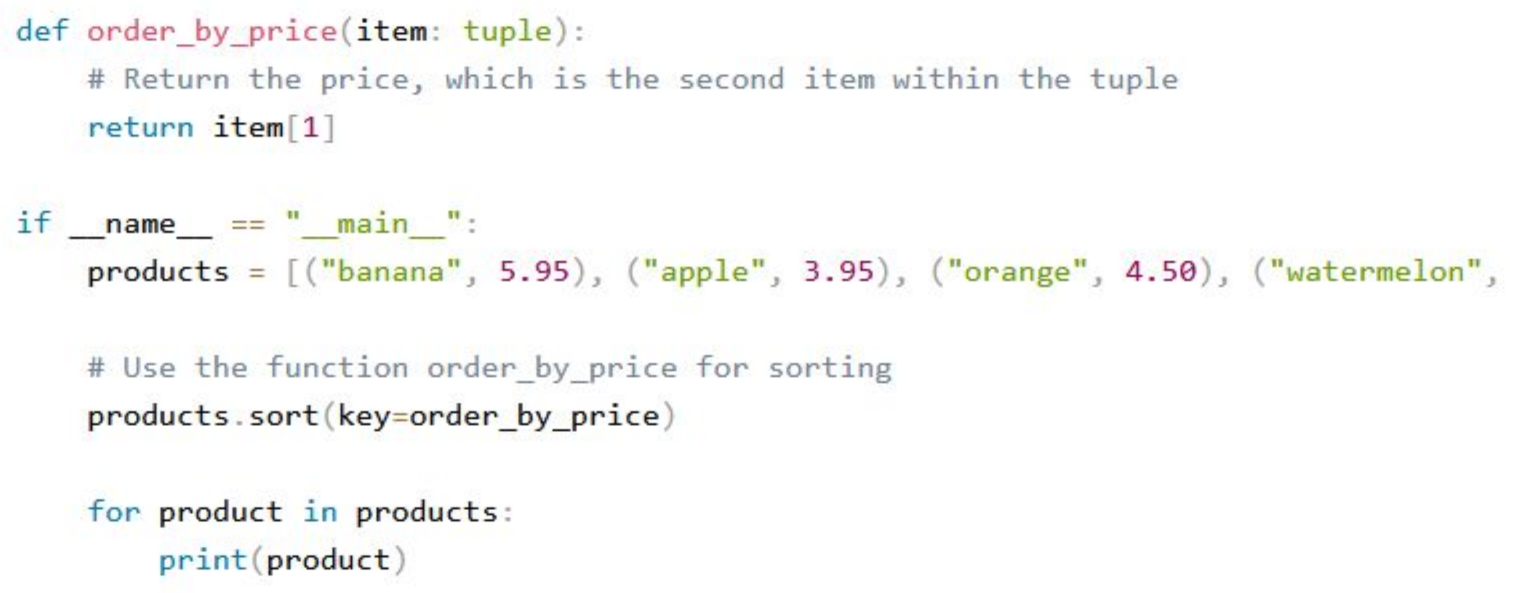
Recursion

Sorting

Problematic if we want to sort something like tuples or our own objects

E.g. tuples are by default sorted based on their first item

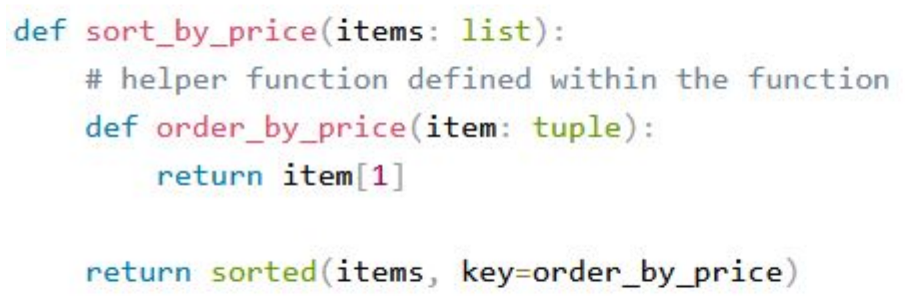
Solution: provide your own value function



```python
def order_by_price(item: tuple):
    # Return the price, which is the second item within the tuple
    return item[1]

if __name__ == "__main__":
    products = [("banana", 5.95), ("apple", 3.95), ("orange", 4.50), ("watermelon",

    # Use the function order_by_price for sorting
    products.sort(key=order_by_price)

    for product in products:
        print(product)
```

Defining functions inside other functions

A "helper function" that is not needed elsewhere can be defined inside other function



```python
def sort_by_price(items: list):
    # helper function defined within the function
    def order_by_price(item: tuple):
        return item[1]

    return sorted(items, key=order_by_price)
```

Lambda-expression

Creates an **anonymous function**

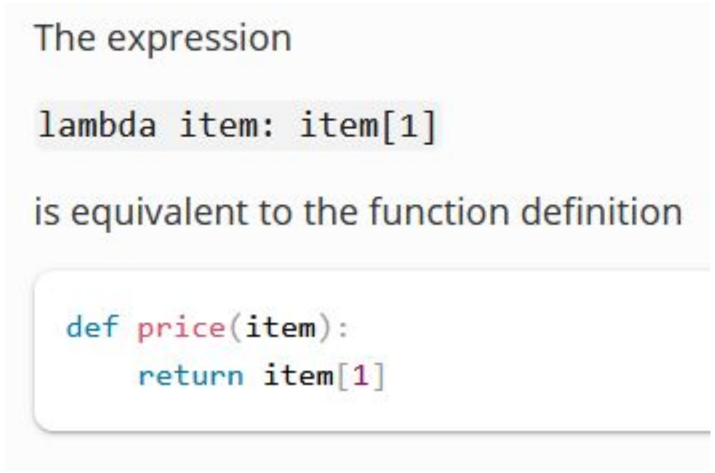
Syntax:

```
lambda <parameters> : <expression>
```

For example

```python
strings = ["Mickey", "Mack", "Marvin", "Minnie", "Merl"]

for word in sorted(strings, key=lambda word: word[-1]):
    print(word)
```

That means that…



The expression

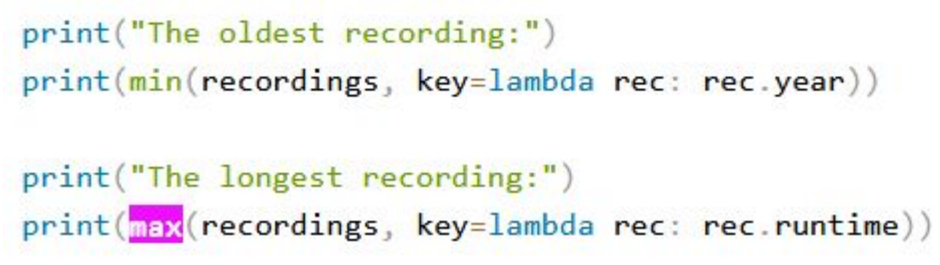
`lambda item: item[1]`

is equivalent to the function definition

```
def price(item):
    return item[1]
```

Min and max

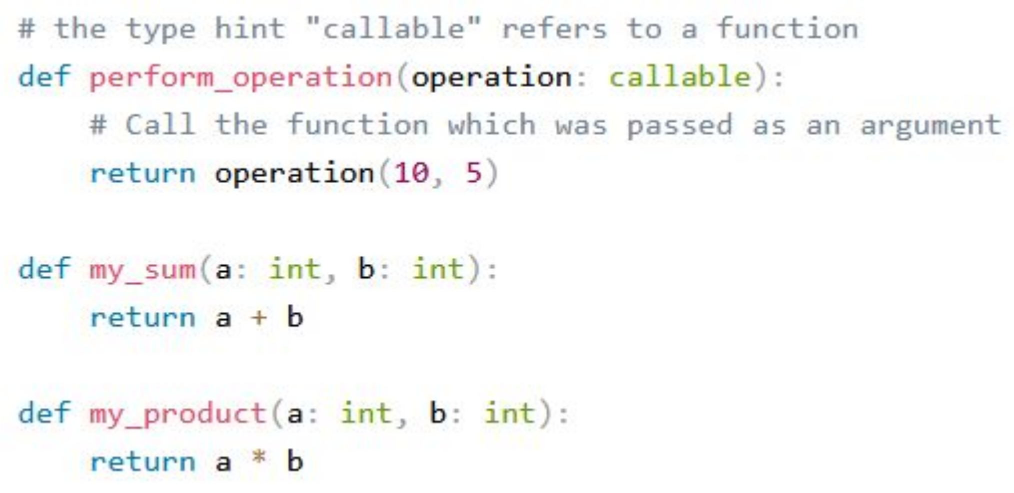
Functions min and max also have an optional **key** parameter



```
print("The oldest recording:")
print(min(recordings, key=lambda rec: rec.year))

print("The longest recording:")
print(max(recordings, key=lambda rec: rec.runtime))
```

Function as an argument

In Python, a function can be passed as an argument:



```python
# the type hint "callable" refers to a function
def perform_operation(operation: callable):
    # Call the function which was passed as an argument
    return operation(10, 5)

def my_sum(a: int, b: int):
    return a + b

def my_product(a: int, b: int):
    return a * b
```

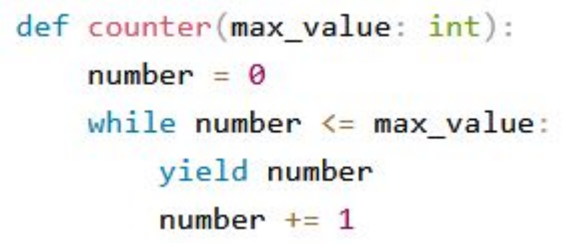
Generators

Sometimes it would be useful to return values from a "series" one at a time without needing to generate the entire list

For this reason, we can use *generator functions*

For example

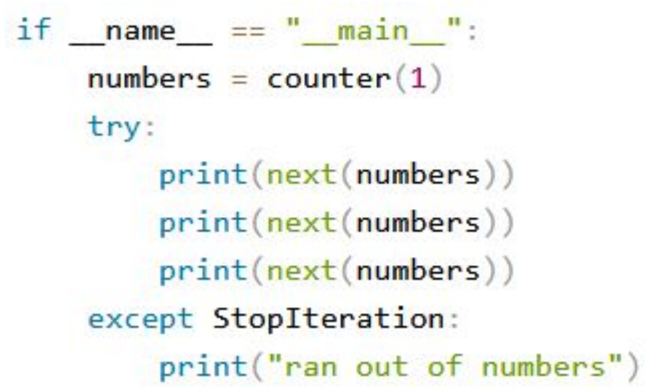
Generator which returns values until maximum



```python
def counter(max_value: int):
    number = 0
    while number <= max_value:
        yield number
        number += 1
```

StopIteration

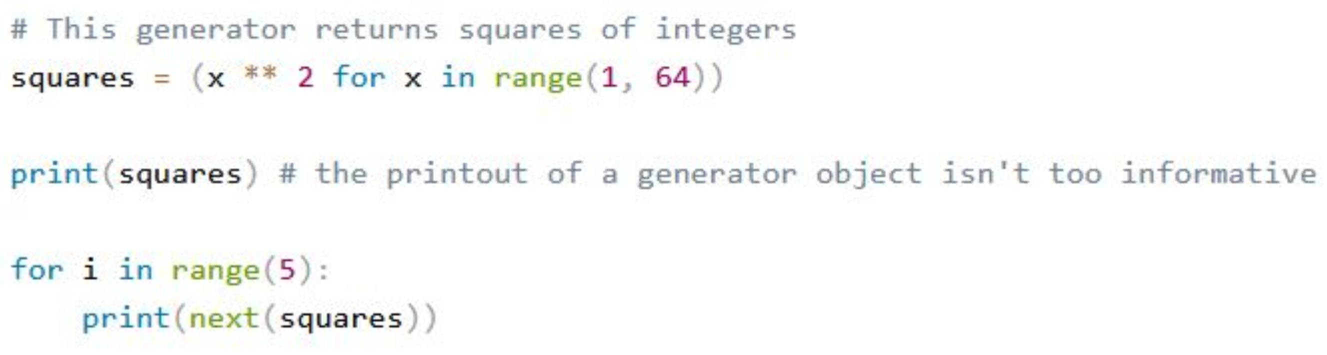
Generator throws a StopIteration event when there are no more values to fetch



```
if __name__ == "__main__":
    numbers = counter(1)
    try:
        print(next(numbers))
        print(next(numbers))
        print(next(numbers))
    except StopIteration:
        print("ran out of numbers")
```

Generator "comprehension"

An alternative syntax for creating a generator with a single expression



```python
# This generator returns squares of integers
squares = (x ** 2 for x in range(1, 64))

print(squares) # the printout of a generator object isn't too informative

for i in range(5):
    print(next(squares))
```

Functional programming

A programming paradigm where the changes in state are avoided

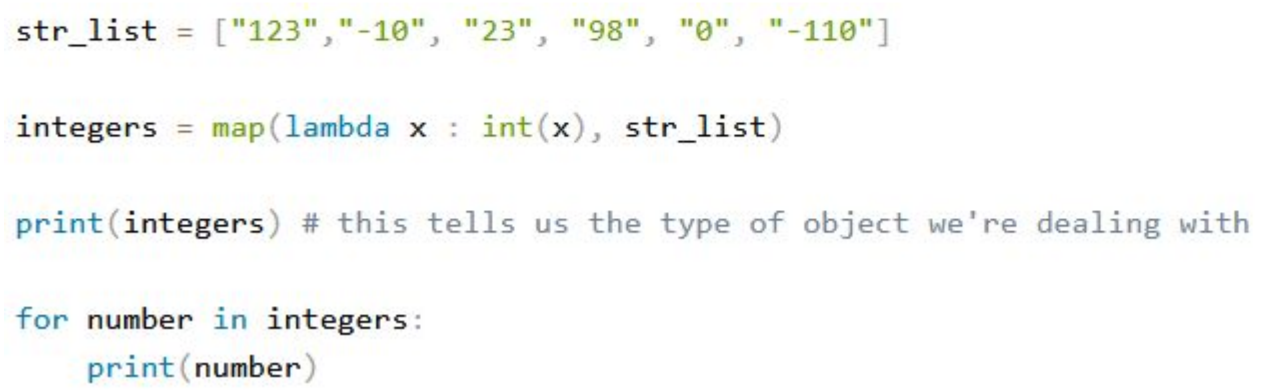
Lambda and expressions are examples of this

Other paradigms:

- Imperative
- Procedural
- Object-oriented

map

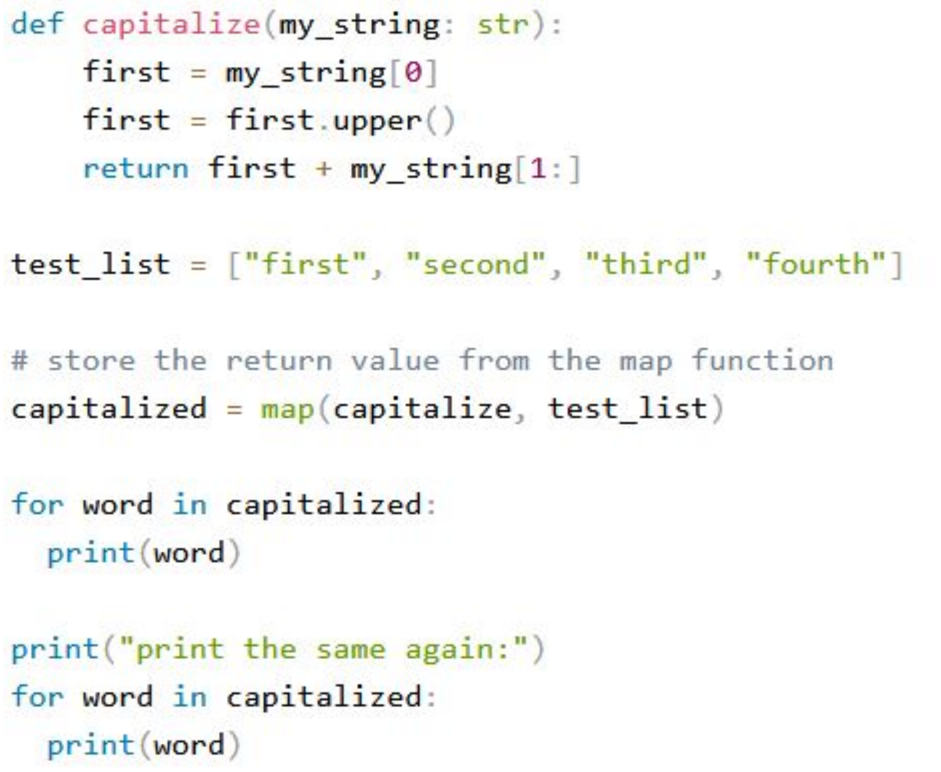
Performs the given operation for all items in the sequence



```python
str_list = ["123","-10", "23", "98", "0", "-110"]

integers = map(lambda x : int(x), str_list)

print(integers) # this tells us the type of object we're dealing with

for number in integers:
    print(number)
```

Return value of map

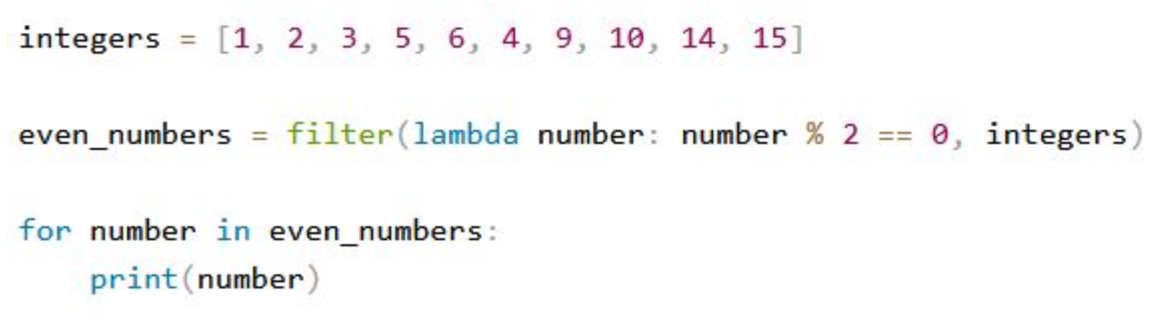
map does not return a list; instead, it returns a sequence which can be iterated once



```python
def capitalize(my_string: str):
    first = my_string[0]
    first = first.upper()
    return first + my_string[1:]

test_list = ["first", "second", "third", "fourth"]

# store the return value from the map function
capitalized = map(capitalize, test_list)

for word in capitalized:
  print(word)

print("print the same again:")
for word in capitalized:
  print(word)
```

filter

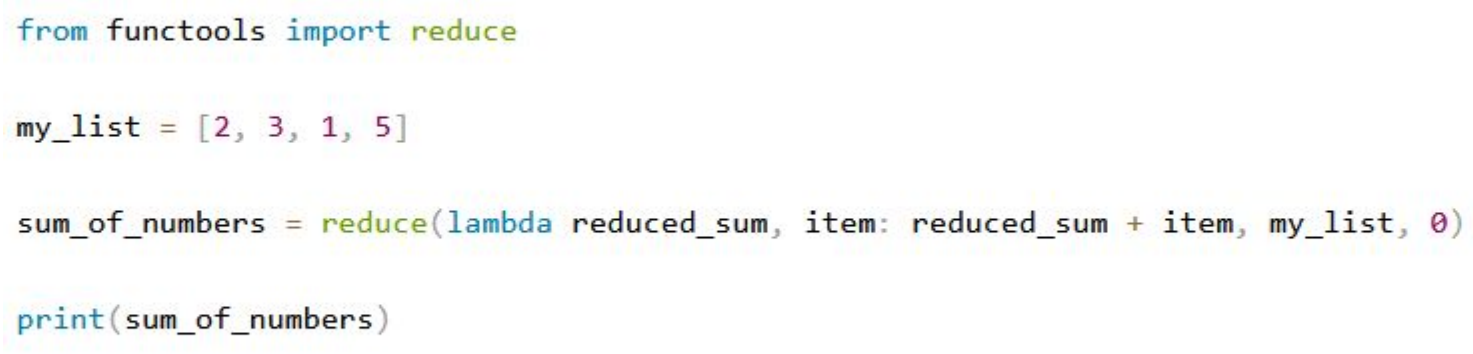
Only selects some of the items in the original sequence based on a condition



```python
integers = [1, 2, 3, 5, 6, 4, 9, 10, 14, 15]

even_numbers = filter(lambda number: number % 2 == 0, integers)

for number in even_numbers:
    print(number)
```

reduce

Reduces the iterable sequence into a single value



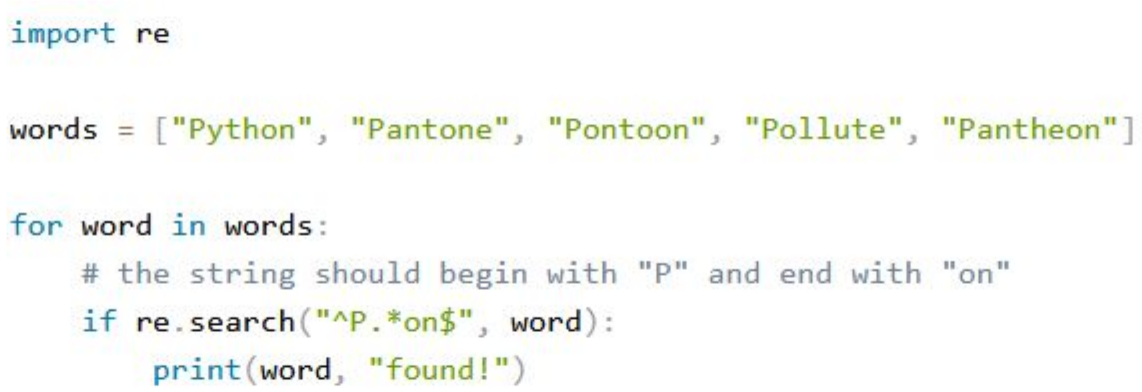
```
from functools import reduce

my_list = [2, 3, 1, 5]

sum_of_numbers = reduce(lambda reduced_sum, item: reduced_sum + item, my_list, 0)

print(sum_of_numbers)
```

Regular Expressions

A "language" for filtering and searching for strings

Own syntax for defining the set of accepted strings

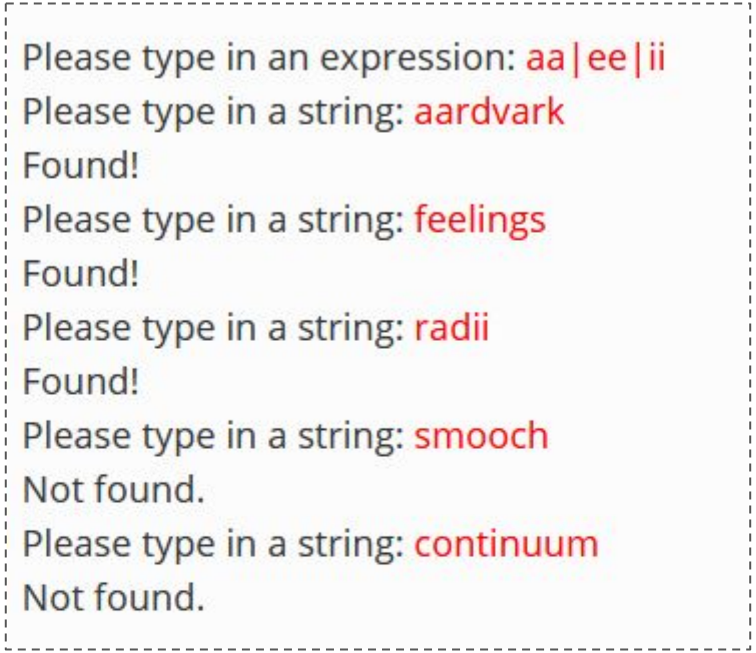
In Python



```python
import re

words = ["Python", "Pantone", "Pontoon", "Pollute", "Pantheon"]

for word in words:
    # the string should begin with "P" and end with "on"
    if re.search("^P.*on$", word):
        print(word, "found!")
```

Rules

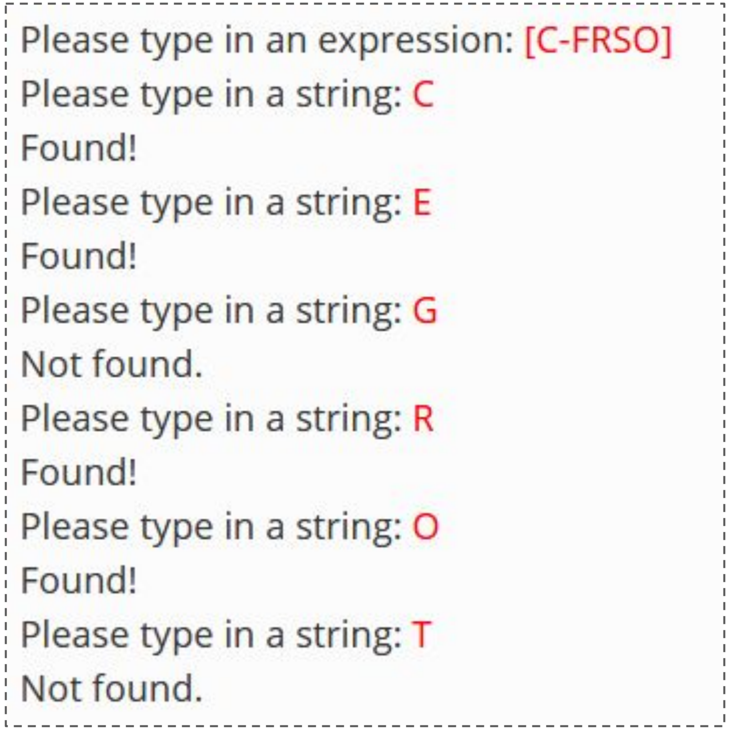
Alternative choices can be defined with a vertical bar



```
Please type in an expression: aa|ee|ii
Please type in a string: aardvark
Found!
Please type in a string: feelings
Found!
Please type in a string: radii
Found!
Please type in a string: smooch
Not found.
Please type in a string: continuum
Not found.
```

Rules (2)

A group of accepted characters (or substrings) is given in square brackets



```
Please type in an expression: [C-FRSO]
Please type in a string: C
Found!
Please type in a string: E
Found!
Please type in a string: G
Not found.
Please type in a string: R
Found!
Please type in a string: O
Found!
Please type in a string: T
Not found.
```

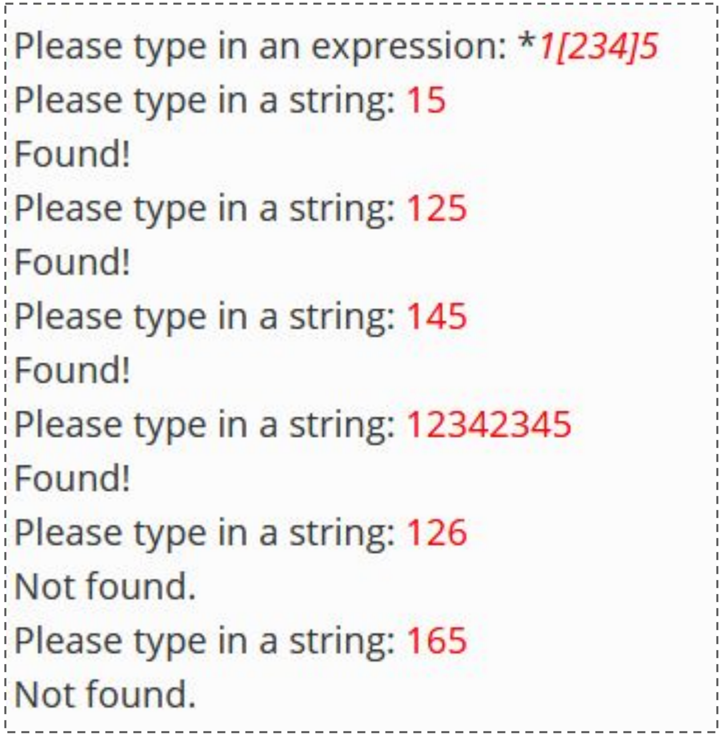
Rules (3)

Number required:

* zero or more

+ one or more

{m} exactly m



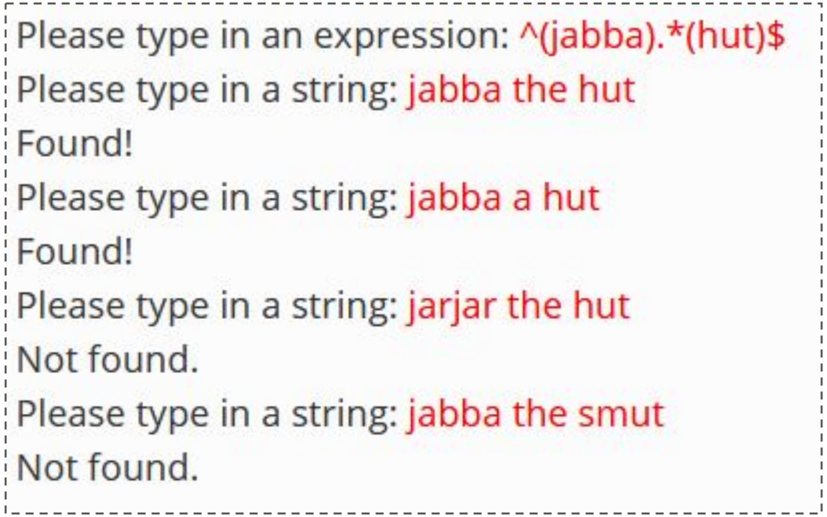
```
Please type in an expression: *1[234]5
Please type in a string: 15
Found!
Please type in a string: 125
Found!
Please type in a string: 145
Found!
Please type in a string: 12342345
Found!
Please type in a string: 126
Not found.
Please type in a string: 165
Not found.
```

Other special characters

Dot denotes any character

^ means that the match must be in the beginning

$ means that the match must be in the end



```
Please type in an expression: ^(jabba).*(hut)$
Please type in a string: jabba the hut
Found!
Please type in a string: jabba a hut
Found!
Please type in a string: jarjar the hut
Not found.
Please type in a string: jabba the smut
Not found.
```

Next Week

One more lecture.

Game programming with Pygame