



Programming 2

Dries Decuyper
Serhat Erdogan
Janne Gilis
Bart Thumas

Functional Programming

AGENDA

- Functional Programming
- Comprehensions
- Higher Order Functions
- Nested Functions



Functional
Programming

Comprehensions

Higher Order
Functions

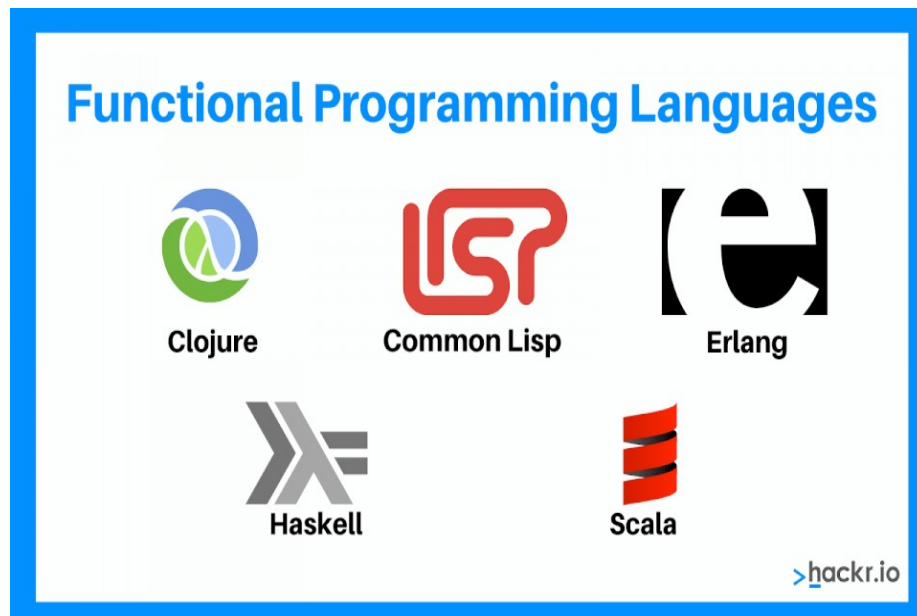
Nested Functions

Functional Programming

- Definition
- Key Concepts



DEFINITION



Functional Programming is a programming paradigm where we can write programs by combining basic building blocks or functions. These functions are pure, meaning that they do not have any side effects. It also supports higher-order functions, which means we can pass functions as arguments to other functions and return functions from functions.

KEY CONCEPTS



```
gim{position:absolute;z-index:999;top:0px;right:0px;#ccc}.gbt1 .gbm{-moz-b
olor:#ccc;display:block;position:absolu
line=5);*opacity:1;*top:-2px;*left:-5px;
width:100%;top:-4px\0/;left:-6px\0/;rig
-moz-inline-box;display:inline-block;fo
0..gimoc{display:block;list-style:none;
play:inline-block;line-height:27px;padd
st(cursor:pointer;display:block;text-de
ition:relative;z-index:1000).gbts{*disp
id).gbtsa(padding-right:9px)#gbz .gbzt,
h(0).gbt4
background:url(
```

- Immutability
- Function Composition
- Deterministic & Pure Functions
- High Order Functions
- Curried Functions
- Functors
- Monads
- [Read More](#)

Functional
Programming

Comprehensions

Higher Order
Functions

Nested Functions

Comprehensions

- List Mapping
- List Filtering
- Set Comprehension
- Dictionary Comprehension
- Flatmap
- Build-In Functions

LIST MAPPING


- Maps each value from an input list to a corresponding value which is stored in an output list.

- List Comprehension
 - Readable way to perform mappings
 - Better performance

```
#list mapping
def double(lst):
    newlist = []
    for value in lst:
        newlist.append(value*2)
    return newlist
```

```
#list mapping using list comprehension
def double_comprehension(lst):
    return [value*2 for value in lst]
```

LIST FILTERING



```
#list filtering
def select_mammals(lst): #list with animal objects
    newlist = []
    for animal in lst:
        if animal.classification == "Mammal":
            newlist.append(animal)
    return newlist

#list filtering using comprehension
def select_mammals_comprehension(lst):
    return [animal for animal in lst if animal.classification == "Mammal"]

#combo mapping (animal.species) and filtering
def select_mammals_species(lst):
    return [animal.species for animal in lst if animal.classification == "Mammal"]
#readability: spread over lines
```


SET AND DICTIONARY COMPREHENSION



- Identical to List Comprehension

```
def double(ns):  
    return {value*2 for value in ns}
```

```
def find_product(product_list, product_code): #list with product objects  
    for product in product_list:  
        if product.code == product_code:  
            return product
```

#By storing them in a dictionary, this search can be done much more efficiently:

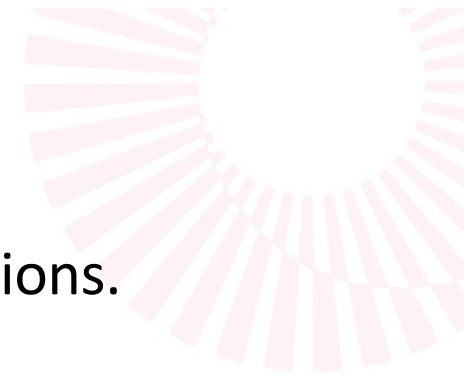
```
def find_product_using_dict(product_list, product_code):  
    product_dict = {product.code: product for product in product_list}  
    return product_dict[product_code]
```

FLATMAP

- Flatten the list of lists into a list

```
[(x,y) for x in range(3)] for y in range(3)] #comprehension in comprehension  
[(x,y) for x in range(3) for y in range(3)] #flattened  
[(x,y,z) for x in range(3) for y in range(3) for z in range(3)] #multiple if's or for's possible
```

BUILD-IN FUNCTIONS



- Built-in functions, often used in combo with comprehensions.
 - Iterable!
 - len, min, max, sum, all, any, zip, enumerate
- The `all()` function returns `True` if all items in an iterable are true, otherwise it returns `False`. If the iterable object is empty, the `all()` function also returns `True`.
- The `any()` function returns `True` if any item in an iterable is true, otherwise it returns `False`. If the iterable object is empty, the `any()` function will return `False`.

BUILD-IN FUNCTIONS

- `zip()` takes two collections and pairs up their first elements, then second elements, etc

```
>>> xs = ['a', 'b', 'c']
>>> ys = [1, 2, 3]
>>> list(zip(xs, ys))
[('a', 1), ('b', 2), ('c', 3)]
```

- `enumerate()` pairs up elements with their index

```
>>> xs = ['a', 'b', 'c']
>>> list(enumerate(xs))
[(0, 'a'), (1, 'b'), (2, 'c')]
```

EXERCISE

- Try the following exercises
- 05-functional-programming
 - 01-comprehensions



Functional
Programming

Comprehensions

Higher Order
Functions

Nested Functions

Higher Order Functions

- Definition
- Smart Parameters
- Generalizing



DEFINITION

- A higher-order function is a function that takes one or more functions as arguments or returns a function as its result.
- One of the benefits of using higher-order functions is the ability to write generalized code that can work with a wide range of inputs.



SMART PARAMETERS



```
def count_animals_by_Sime(animals):  
    #animals is a list with Animal objects  
    #Sime is the name of a caretaker  
    return len([animal for animal in animals if animal.caretaker == "Sime"])  
  
#Not good to hard code the value, seperate variable:  
def count_animals_by_Sime(animals):  
    caretaker = "Sime" ##  
    return len([animal for animal in animals if animal.caretaker == caretaker])  
  
#More generalized function by using variable as parameter of function  
def count_animals_by_caretaker(animals, caretaker):  
    return len([animal for animal in animals if animal.caretaker == caretaker])
```

But we also want to know other counts!

SMART PARAMETERS

```
#Function to count animals in certain area of the zoo
def count_animals_in_area(animals, area):
    return len([animal for animal in animals if animal.area == area])

#Function to count animals older than a certain age.
#Copy paste code above and adapt to show repetition
def count_animals_above_age(animals, age):
    return len([animal for animal in animals if animal.age >= age])

#Animals younger than a certain age
#Animals with healthcare issues
#....
```

Those functions have a lot in common > Duplication !

SMART PARAMETERS



Let's move the condition in another function!

```
#Putting the condition in separate function (like we stored value in variable (caretaker))
def is_right_caretaker(animal):
    return animal.caretaker == "Sime"

#using that function for the if-statement
def count_animals_by_Sime(animals):
    return len([animal for animal in animals if is_right_caretaker(animal)])

#When GENERALIZING count_animals_by_Sime --> count_animals_by_caretaker
# next we turned the variable into a parameter of the function.
# Works with the function as well

def count_animals_by_caretaker(animals, is_right_caretaker):
    return len([animal for animal in animals if is_right_caretaker(animal)])
#above we directly called the function
#Here passed as parameter, parameter name here just the same
```

GENERALIZING

```
def caretaker_is_Sime(animal):
    return animal.caretaker == "Sime"

def caretaker_is_Mickey(animal):
    return animal.caretaker == "Mickey"

count_animals_by_caretaker(animals, caretaker_is_Sime)
count_animals_by_caretaker(animals, caretaker_is_Mickey)
#count_animals_by_caretaker function not specific for caretakers only
#returns a list of animals who meet the condition

def lives_in_area_a(animal):
    return animal.area == "area_a"

count_animals_by_caretaker(animals, lives_in_area_a)

# misleading name, rename the function
def count_animals_(animals, condition_function):
    return len([animal for animal in animals if condition_function(animal)])

#This function can replace all functions we discussed
```



GENERALIZING

- Further generalizing: count function, count any kind of object based on condition (input list of objects)

```
def count(xs, should_be_counted):  
    return len([x for x in xs if should_be_counted(x)])
```

EXERCISE

- Try the following exercises
- 05-functional-programming
 - 02-higher-order-functions



Functional
Programming

Comprehensions

Higher Order
Functions

Nested Functions

Nested Functions



NESTED FUNCTIONS

- Functions in Functions
- Can access variables as if “a local variable”

```
#generalized count function
def count(xs, should_be_counted):
    return len([x for x in xs if should_be_counted(x)])

def caretaker_is_sime(animal):
    return animal.caretaker == "Sime"

#relying on count function
def count_animals_by_sime(animals):
    return count(animals, caretaker_is_sime)
#created little function caretaker_is_sime to be passed to count
#clumsy don't want function for every caretaker

#Let's write function count_animals_by_caretaker(animals, caretaker)
# relying on count
# --> we pass the caretaker as argument as well
# and refer to it in the condition-function...
def caretaker_is(animal):
    return animal.caretaker == caretaker
#... which is not possible as it is a local variable

def count_animals_by_caretaker(animals, caretaker):
    return count(animals, caretaker_is)

#param only visible inside the function so...
def count_animals_by_caretaker(animals, caretaker):
    def caretaker_is(animal):
        return animal.caretaker == caretaker

    return count(animals, caretaker_is)
```

NESTED FUNCTIONS



```
#generalized count function
def count(xs, should_be_counted):
    return len([x for x in xs if should_be_counted(x)])

#param only visible inside the function so...
def count_animals_by_caretaker(animals, caretaker):
    def caretaker_is(animal):
        return animal.caretaker == caretaker

    return count(animals, caretaker_is)
```

```
graph TD
    A((xs)) --> B((animals))
    C((should_be_counted)) --> D((caretaker_is))
    E((animals)) --> F((animals))
    G((caretaker_is)) --> H((caretaker_is))
```


NESTED FUNCTIONS

```
#generalized count function
def count(xs, should_be_counted):
    return len([x for x in xs if should_be_counted(x)])

#param only visible inside the function so...
def count_animals_by_caretaker(animals, caretaker):
    def caretaker_is(animal):
        return animal.caretaker == caretaker

    return count(animals, caretaker_is)
```

The diagram illustrates the execution flow of the nested functions. A red circle highlights the `count` function definition. A red arrow points from this circle to the `count` function call within `count_animals_by_caretaker`. Another red circle highlights the `animals` parameter in the `count` call, with a red arrow pointing to the `animals` parameter in the `count_animals_by_caretaker` definition. A third red circle highlights the `caretaker_is` function definition, with a red arrow pointing to its call within `count_animals_by_caretaker`. A blue circle highlights the `should_be_counted(x)` expression in the `count` function's list comprehension, with a blue arrow pointing to the `caretaker_is` function call, indicating that `caretaker_is` is passed as the `should_be_counted` argument.

NESTED FUNCTIONS

The diagram illustrates the execution flow of the provided Python code. It features two functions: a top-level `count` function and a nested `count_animals_by_caretaker` function. The `count` function is annotated with a red circle around its first parameter `xs` and a blue oval around its second parameter `should_be_counted`. The nested function `count_animals_by_caretaker` has a red circle around its first parameter `animals` and a blue oval around its second parameter `caretaker`. Inside this nested function, there is another nested function `caretaker_is` with a blue oval around its parameter `animal`. The return statement of `count_animals_by_caretaker` calls the `count` function, passing `animals` and `caretaker_is` as arguments. Arrows indicate the flow of data: a red arrow points from the `animals` parameter of `count_animals_by_caretaker` to the `xs` parameter of `count`; a blue arrow points from the `caretaker_is` function object to the `should_be_counted` parameter of `count`. Additionally, a blue oval highlights the `should_be_counted(x)` expression within the list comprehension of the `count` function, with a blue arrow pointing to the `x` parameter of this inner function call.

```
#generalized count function
def count(xs, should_be_counted):
    return len([x for x in xs if should_be_counted(x)])

#param only visible inside the function so...
def count_animals_by_caretaker(animals, caretaker):
    def caretaker_is(animal):
        return animal.caretaker == caretaker

    return count(animals, caretaker_is)
```

EXERCISE

- Try the following exercises
- 05-functional-programming
 - 03-nested-functions

