Higher order functions and SQL

TEAM INFDEV

Hogeschool Rotterdam
Rotterdam, Netherlands

Introduction

Introduction

Motivation

- Sometimes simple functions are not flexible enough
- We might have similar algorithms that are "not quite" the same
- For example, consider adding or multiplying all elements of a list together
  - **"Consider" here actually means do it on paper and then a volunteer comes implement it at the lecturer's PC**

Higher order function

Idea

- Functions may also take and return other functions as parameters
  - These are then called **higher order functions** (HOF's)[a]
- This lets us specify a function where some instructions are not fixed
- By passing other functions as parameters we literally create "customizable algorithms"

a

Higher order function

Idea

- Functions may also take and return other functions as parameters
  - These are then called **higher order functions** (HOF's)[a]
- This lets us specify a function where some instructions are not fixed
- By passing other functions as parameters we literally create "customizable algorithms"

[a] **Higher order** because parameters are not concrete values but rather computations, which are higher wrt the floors of the Ivory Tower

Higher order function

Example

- As an example, consider the case of combining two values together
- We do not care how, as long as they are combined according to some criterion
- The criterion is given as an input function

HOF's


```
def combine(op,x,y):
  return op(x,y)
```

Example

- What do we know about x and y?
- Do we even care?

Higher order function

Example

- A function such as `combine` can be used by providing another function as the first parameter
- As long as the function will work correctly on the second and third parameters

HOF's


```
def combine(op,x,y):
  return op(x,y)

def plus(x,y): return x + y
def times(x,y): return x * y
def minus(x,y): return x - y

print(combine(plus, 10, 20))
print(combine(times, 10, 20))
print(combine(minus, 10, 20))
```

Example

- **What does this code do?**

HOF's


```
def combine(op,x,y):
  return op(x,y)

def plus(x,y): return x + y
def times(x,y): return x * y
def minus(x,y): return x - y

print(combine(plus, 10, 20))
print(combine(times, 10, 20))
print(combine(minus, 10, 20))
```

Example

- **What does this code do?**
- Prints 30, 200, −10

Higher order function

Example

- We can use `combine` on any data types we want
- For example, strings

HOF's


```
def combine(op,x,y):
  return op(x,y)

def plus(x,y): return x + y
def times(x,y): return x * y
def minus(x,y): return x - y

print(combine(plus, "10", "20"))
print(combine(times, 10, 20))
print(combine(minus, 10, 20))
```

Example

- **What does this code do?**

HOF's

```
def combine(op,x,y):
  return op(x,y)

def plus(x,y): return x + y
def times(x,y): return x * y
def minus(x,y): return x - y

print(combine(plus, "10", "20"))
print(combine(times, 10, 20))
print(combine(minus, 10, 20))
```

Example

- **What does this code do?**
- Prints 1020, 200, −10

HOF's

```
def combine(op,x,y):
  return op(x,y)

def plus(x,y): return x + y
def times(x,y): return x * y
def minus(x,y): return x - y

print(combine(plus, "10", "20"))
print(combine(times, 10, 20))
print(combine(minus, 10, 20))
```

What do stack and heap look like from inside a call to `combine`?

HOF's

```
def combine(op,x,y):
  return op(x,y)

def plus(x,y): return x + y
def times(x,y): return x * y
def minus(x,y): return x - y

print(combine(plus, "10", "20"))
print(combine(times, 10, 20))
print(combine(minus, 10, 20))
```

What do stack and heap look like from inside a call to `combine`?

S

PC	combine	PC	op	x	y
8	nil	2	ref(plus)	"10"	"20"

H

or

S

PC	combine	PC	op	x	y
8	nil	2	ref(times)	10	20

H

Higher order function

Lambda-syntax function definition

- Defining functions such as `plus`, `times`, and `minus` is cumbersome
- After all, we already have symbols for them: `(+)`, `(*)`, and `(-)`
- Repetition and duplication of code is never good

Higher order function

Lambda-syntax function definition

- Python (version at least 3) offers facilities for the inline definition of short functions
- The syntax fits one line and requires no newlines
- lambda <<paramters>>: <<result>>
  - <<parameters>> is a list of comma-separated parameters
  - <<result>> is the expression that is returned
- For example: lambda x,y: x+y

HOF's


```
def combine(op,x,y):
  return op(x,y)

print(combine((lambda x,y: x+y), "10", "20"))
print(combine((lambda x,y: x*y), 10, 20))
print(combine((lambda x,y: x-y), 10, 20))
```

Lambda-syntax function definition

- **What does this code do?**

```
def combine(op,x,y):
  return op(x,y)

print(combine((lambda x,y: x+y), "10", "20"))
print(combine((lambda x,y: x*y), 10, 20))
print(combine((lambda x,y: x-y), 10, 20))
```

Lambda-syntax function definition

- **What does this code do?**
- Prints 1020, 200, −10
- Does not require the extra function definitions

HOF's


```
def combine(op,x,y):
  return op(x,y)

print(combine((lambda x,y: x+y), "10", "20"))
print(combine((lambda x,y: x*y), 10, 20))
print(combine((lambda x,y: x-y), 10, 20))
```

What do stack and heap look like from inside a call to `combine`?

```
def combine(op,x,y):
  return op(x,y)

print(combine((lambda x,y: x+y), "10", "20"))
print(combine((lambda x,y: x*y), 10, 20))
print(combine((lambda x,y: x-y), 10, 20))
```

What do stack and heap look like from inside a call to `combine`?

S	PC	combine	PC	op	x	y
	4	nil	2	ref(0)	"10"	"20"

H	0
	lambda x,y: x+y

or

S	PC	combine	PC	op	x	y
	5	nil	2	ref(1)	10	20

H	0	1
	lambda x,y: x+y	lambda x,y: x*y

Higher order function

Lambda-syntax function definition

- We can also return a function from a function
- For example, to dynamically choose an operation
- This makes code very expressive and flexible, but also potentially much harder to read
- Use with caution!

HOF's


```
def combine(op,x,y):
  return op(x,y)

def choose_operation():
  i = input("Choose an operation between +, -, or *")
  if i == "+":
    return lambda x,y: x+y
  elif i == "-":
    return lambda x,y: x-y
  else:
    return lambda x,y: x*y
print(combine(choose_operation(), 10, 20))
```

Lambda-syntax function definition

- **What does this code do?**

HOF's

```
def combine(op,x,y):
  return op(x,y)

def choose_operation():
  i = input("Choose␣an␣operation␣between␣+,␣-,␣or␣*")
  if i == "+":
    return lambda x,y: x+y
  elif i == "-":
    return lambda x,y: x-y
  else:
    return lambda x,y: x*y
print(combine(choose_operation(), 10, 20))
```

Lambda-syntax function definition

- **What does this code do?**
- Chooses the function based on `input` that will `combine` 10 and 20

HOF's


```
def combine(op,x,y):
  return op(x,y)

def choose_operation():
  i = input("Choose an operation between +, -, or *")
  if i == "+":
    return lambda x,y: x+y
  elif i == "-":
    return lambda x,y: x-y
  else:
    return lambda x,y: x*y
print(combine(choose_operation(), 10, 20))
```

What do stack and heap look like after `choose_operation` terminates?

HOF's


```
def combine(op,x,y):
  return op(x,y)

def choose_operation():
  i = input("Choose␣an␣operation␣between␣+,␣-,␣or␣*")
  if i == "+":
    return lambda x,y: x+y
  elif i == "-":
    return lambda x,y: x-y
  else:
    return lambda x,y: x*y
print(combine(choose_operation(), 10, 20))
```

What do stack and heap look like after `choose_operation` terminates?

S

PC	choose_operation
12	ref(0)

H

0
lambda x,y: x+y

List HOF's

List HOF's

Introduction

- Consider our (now well-known) list implementation
- `Empty` and `Node` classes
- `IsEmpty`, `Head`, `Tail` methods

List definition


```
class Empty:
  def __init__(self):
    self.IsEmpty = True
Empty = Empty()

class Node:
  def __init__(self, x, xs):
    self.IsEmpty = False
    self.Head = x
    self.Tail = xs

def printList(l):
  if(l.IsEmpty):
    return Empty
  else:
    print(l.Head)
    printList(l.Tail)
```

List HOF's

Fundamental operations on lists

- What are the **fundamental things** we wish to do with a list?

List HOF's

Fundamental operations on lists

- What are the **fundamental things** we wish to do with a list?
- **Transform** all its elements: $N \rightarrow N$

Fundamental operations on lists

- What are the **fundamental things** we wish to do with a list?
- **Transform** all its elements: $N \rightarrow N$
- **Filter** some of its elements: $N \rightarrow M, M \leq N$

List HOF's

Fundamental operations on lists

- What are the **fundamental things** we wish to do with a list?
- **Transform** all its elements: $N \rightarrow N$
- **Filter** some of its elements: $N \rightarrow M, M \leq N$
- **Fold** its elements into a single value: $N \rightarrow 1$

Transforming a list


```
def map(l, f):
  if(l.IsEmpty):
    return Empty
  else:
    return Node(f(l.Head), map(l.Tail, f))

printList(map(Node(1, Node(2, Node(3, Node(4, Empty)))), lambda x: x + 1))
```

Fundamental operations on lists

- What does the code above print?

Transforming a list


```
def map(l, f):
  if(l.IsEmpty):
    return Empty
  else:
    return Node(f(l.Head), map(l.Tail, f))

printList(map(Node(1, Node(2, Node(3, Node(4, Empty)))), lambda x: x + 1))
```

Fundamental operations on lists

- What does the code above print?
- 2, 3, 4, 5

Transforming a list


```
def map(l, f):
  if(l.IsEmpty):
    return Empty
  else:
    return Node(f(l.Head), map(l.Tail, f))

printList(map(Node(1, Node(2, Node(3, Node(4, Empty)))), lambda x: x * 2))
```

Fundamental operations on lists

- What does the code above print?

Transforming a list


```
def map(l, f):
  if(l.IsEmpty):
    return Empty
  else:
    return Node(f(l.Head), map(l.Tail, f))

printList(map(Node(1, Node(2, Node(3, Node(4, Empty)))), lambda x: x * 2))
```

Fundamental operations on lists

- What does the code above print?
- `2, 4, 6, 8`

Filtering a list


```
def filter(l, p):
  if(l.IsEmpty):
    return Empty
  else:
    if p(l.Head):
      return Node(l.Head, filter(l.Tail, p))
    else:
      return filter(l.Tail, p)

printList(filter(Node(1, Node(2, Node(3, Node(4, Empty)))), lambda x: x \% 2
      == 0))
```

Fundamental operations on lists

- What does the code above print?

Filtering a list


```
def filter(l, p):
  if(l.IsEmpty):
    return Empty
  else:
    if p(l.Head):
      return Node(l.Head, filter(l.Tail, p))
    else:
      return filter(l.Tail, p)

printList(filter(Node(1, Node(2, Node(3, Node(4, Empty)))), lambda x: x \% 2
      == 0))
```

Fundamental operations on lists

- What does the code above print?
- 2, 4

Folding a list


```
def fold(l, f, z):
  if(l.IsEmpty):
    return z
  else:
    return f(l.Head, fold(l.Tail, f, z))

print(fold(Node(1, Node(2, Node(3, Node(4, Empty)))), lambda x, y: x + y, 0)
     )
```

Fundamental operations on lists

- What does the code above print?

Folding a list

```
def fold(l, f, z):
  if(l.IsEmpty):
    return z
  else:
    return f(l.Head, fold(l.Tail, f, z))

print(fold(Node(1, Node(2, Node(3, Node(4, Empty)))), lambda x, y: x + y, 0)
    )
```

Fundamental operations on lists

- What does the code above print?
- 10

List HOF's

Using HOF's

- We can perform almost anything we need to do no lists with `map`, `filter`, and `fold`
- Some complex algorithm cannot be implemented relying on unbounded recursion (where we cannot estimate the maximum number of steps)
- This happens because `map`, `filter`, and `fold` will always terminate (if the input function terminates)
- Still, they are quite powerful in their capabilities

List HOF's

Using HOF's

- `map` is very obvious: transform elements
  - `map(cars, drive)`
  - `map(planes, fly)`
  - `map(bikes, pedal)`
  - ...

List HOF's

Using HOF's

- `filter` is also very obvious: remove useless elements
  - `filter(cars, arrived)`
  - `filter(planes, landed)`
  - `filter(bikes, crashed)`
  - ...

List HOF's

Using HOF's

- `fold` is much more complex
- Recall that it folds a list into a single value $N \rightarrow 1$
  - `fold(l, lambda x,l:  l + 1, 0) = ?`

Using HOF's

- `fold` is much more complex
- Recall that it folds a list into a single value $N \rightarrow 1$
  - `fold(l, lambda x,l:  l + 1, 0) = ?` length of `l`
  - `fold(l, max, float('-inf')) = ?`

List HOF's

Using HOF's

- `fold` is much more complex
- Recall that it folds a list into a single value $N \rightarrow 1$
  - `fold(l, lambda x,l:  l + 1, 0) = ?` length of `l`
  - `fold(l, max, float('-inf')) = ?` max of `l`
  - `fold(l, min, float('inf')) = ?`

List HOF's

Using HOF's

- `fold` is much more complex
- Recall that it folds a list into a single value $N \to 1$
  - `fold(l, lambda x,l:  l + 1, 0) = ?` length of `l`
  - `fold(l, max, float('-inf')) = ?` max of `l`
  - `fold(l, min, float('inf')) = ?` min of `l`
  - `fold(cars, closerToPlayer, None) = ?`

List HOF's

Using HOF's

- `fold` is much more complex
- Recall that it folds a list into a single value $N \rightarrow 1$
  - `fold(l, lambda x,l:  l + 1, 0) = ?` length of `l`
  - `fold(l, max, float('-inf')) = ?` max of `l`
  - `fold(l, min, float('inf')) = ?` min of `l`
  - `fold(cars, closerToPlayer, None) = ? closest car to player`
  - ...

List HOF's

Folding to lists

- `fold` can return a value of an arbitrary type
- **Also a list?**

List HOF's

Folding to lists

- `fold` can return a value of an arbitrary type
- **Also a list?** Yes!

Folding to lists


```
printList(
  fold(
    Node(1, Node(2, Node(3, Node(4, Empty)))),
    lambda x, y: Node(x+1,y),
    Empty))
```

Folding to lists

- What does the code above print?

Folding to lists


```
printList(
  fold(
    Node(1, Node(2, Node(3, Node(4, Empty)))),
    lambda x, y: Node(x+1,y),
    Empty))
```

Folding to lists

- What does the code above print?
- `2, 3, 4, 5`
- What does it look like?

Folding to lists

```
printList(
  fold(
    Node(1, Node(2, Node(3, Node(4, Empty)))),
    lambda x, y: Node(x+1,y),
    Empty))
```

Folding to lists

- What does the code above print?
- `2, 3, 4, 5`
- What does it look like?
- **A map!**

List HOF's

Combine list HOF's

- We can clearly combine `map`, `filter`, and `fold`
- For example, we could say `filter(map(l, f), p)` that applies a `map` first and a `filter` second
  - `filter(map(cars, drive), arrived) = ?`

List HOF's

Combine list HOF's

- We can clearly combine `map`, `filter`, and `fold`
- For example, we could say `filter(map(l, f), p)` that applies a `map` first and a `filter` second
  - `filter(map(cars, drive), arrived) = ?` updated cars that have not yet arrived

SQL vs list HOF's

SQL vs list HOF's

Introduction

- We will now explore the differences and similarities between SQL and Python list HOF's
- SQL statements translated to Python HOF's
- Python HOF's translated to SQL statements

SQL vs list HOF's

SELECT

- Consider a simple SQL query
- `SELECT f(x) FROM l`
- **What are** `f`, `x`, **and** `l`?

`l` is?

SQL vs list HOF's

SELECT

- Consider a simple SQL query
- `SELECT f(x) FROM l`
- **What are** `f`, `x`, **and** `l`?

  `l` is? a table
  `x` is?

SQL vs list HOF's

SELECT

- Consider a simple SQL query
- `SELECT f(x) FROM l`
- **What are** `f`, `x`, **and** `l`?

  `l` is? a table
  `x` is? an entry from the table
  `f` is?

SQL vs list HOF's

SELECT

- Consider a simple SQL query
- `SELECT f(x) FROM l`
- **What are** `f`, `x`, **and** `l`?

`l`	is? a table
`x`	is? an entry from the table
`f`	is? a transformation of the entries of the table
the query	returns? all elements of `l` transformed by `f`

SQL vs list HOF's

SELECT

- Consider a simple call to `map`
- `map(l, lambda x: f(x))`
- **What are** `f`, `x`, **and** `l`?

`l` is?

SQL vs list HOF's

SELECT

- Consider a simple call to `map`
- `map(l, lambda x: f(x))`
- **What are** `f`, `x`, **and** `l`?

  `l` is? a list

  `x` is?

SQL vs list HOF's

SELECT

- Consider a simple call to `map`
- `map(l, lambda x:  f(x))`
- **What are** `f`, `x`, **and** `l`?

  `l` is? a list
  `x` is? an element of the list
  `f` is?

SQL vs list HOF's

SELECT

- Consider a simple call to `map`
- `map(l, lambda x: f(x))`
- **What are** `f`, `x`, **and** `l`?

`l`	is? a list
`x`	is? an element of the list
`f`	is? a transformation of the elements of the list
the call	returns? all elements of `l` transformed by `f`

SQL vs list HOF's

SELECT

Domain	**Code**	`l`	`x`	`f`	**return**
SQL	`SELECT f(x) FROM l`	table	entry of `l`	transformation of `x`	all `l` transformed by `f`
Python	`map(l, lambda x: f(x))`	list	element of `l`	transformation of `x`	all `l` transformed by `f`
Logic					

SQL vs list HOF's

SELECT

Domain	**Code**	`l`	`x`	`f`	**return**
SQL	`SELECT f(x) FROM l`	table	entry of `l`	transformation of `x`	all `l` transformed by `f`
Python	`map(l, lambda x: f(x))`	list	element of `l`	transformation of `x`	all `l` transformed by `f`
Logic	$\{f(x) \vert x \in l\}$	set	element of `l`	function of `x`	all `l` transformed by `f`

SQL vs list HOF's

WHERE

- Consider now a restriction
- `SELECT * FROM l WHERE p(x)`
- **What are** `p`, `x`, **and** `l`?

  `l` is?

SQL vs list HOF's

WHERE

- Consider now a restriction
- `SELECT * FROM l WHERE p(x)`
- **What are** `p`, `x`, **and** `l`?

  `l` is? a table

  `x` is?

SQL vs list HOF's

WHERE

- Consider now a restriction
- `SELECT * FROM l WHERE p(x)`
- **What are** `p`, `x`, **and** `l`?

  `l` is? a table

  `x` is? an entry from the table

  `p` is?

SQL vs list HOF's

WHERE

- Consider now a restriction
- `SELECT * FROM l WHERE p(x)`
- **What are** `p`, `x`, **and** `l`?

`l`	is? a table
`x`	is? an entry from the table
`p`	is? a condition on the entries of the table
the query	returns? all elements of `l` satisfying `p`

- **What does this correspond to in Python?**

WHERE

- Let's use a `filter`!
- `filter(l, lambda x: p(x))`
- **What are** `p`, `x`, **and** `l`?

`l` is?

SQL vs list HOF's

WHERE

- Let's use a `filter`!
- `filter(l, lambda x:  p(x))`
- **What are** `p`, `x`, **and** `l`?

  `l` is? a list

  `x` is?

SQL vs list HOF's

WHERE

- Let's use a `filter`!
- `filter(l, lambda x:  p(x))`
- **What are** `p`, `x`, **and** `l`?

  `l` is? a list

  `x` is? an element of the list

  `f` is?

SQL vs list HOF's

WHERE

- Let's use a `filter`!
- `filter(l, lambda x:  p(x))`
- **What are** `p`, `x`, **and** `l`?

`l`	is?	a list
`x`	is?	an element of the list
`f`	is?	a condition on the elements of the list
the call	returns?	all elements of `l` satisfying `f`

SQL vs list HOF's

WHERE

Domain	Code	l	x	f	return
SQL	`SELECT * FROM l WHERE p(x)`	table	entry of `l`	condition on `x`	all `l` satisfying `p`
Python	`filter(l, lambda x: p(x))`	list	element of `l`	condition on `x`	all `l` satisfying `p`
Logic					

SQL vs list HOF's

WHERE

Domain	**Code**	`l`	`x`	`f`	**return**
SQL	`SELECT * FROM l WHERE p(x)`	table	entry of `l`	condition on `x`	all `l` satisfying `p`
Python	`filter(l, lambda x: p(x))`	list	element of `l`	condition on `x`	all `l` satisfying `p`
Logic	$\{x \mid x \in l \land p(x)\}$	set	element of `l`	predicate on `x`	`l` restricted to/by `f`

SQL vs list HOF's

AGGREGATE

- Consider now an aggregation
- `SELECT COUNT(*) FROM l`

  the query returns? the number of elements of `l`
- **What does this correspond to in Python?**

SQL vs list HOF's

AGGREGATE

- Let's use a `fold`!
- `fold(l, (lambda x,c: c+1), 0)`

  the call returns? the number of elements of `l`

SQL vs list HOF's

AGGREGATE

Domain	Code	l	return
SQL	`SELECT COUNT(*) FROM l`	table	number of entries of l
Python	`fold(l, lambda x,c: c+1, 0)`	list	number of elements of l
Logic			

SQL vs list HOF's

AGGREGATE

Domain	Code	l	return
SQL	`SELECT COUNT(*) FROM l`	table	number of entries of l
Python	`fold(l, lambda x,c: c+1, 0)`	list	number of elements of l
Logic	$(\| + 1 \|) l$	set	size of l

SQL vs list HOF's

General considerations

- There is no real conceptual difference between SQL and list HOF's
- The mapping is quite straightforward

SQL vs list HOF's

General considerations

Concept	SQL	HOF's
element transformation	SELECT	map
element removal	WHERE	filter
element folding	SUM, COUNT, AVG, ...	fold
cartesian product	JOIN	nesting of HOF's[a]

[a] A filter within a map is a basic join.

Conclusion

Conclusion

Lecture topics

- Often, user code needs to perform operations that are similar to each other
- Through the mechanism of function definition, we can recycle code
- Functions can encode algorithms in many way
  - Simple code abstractions to avoid repetition
  - Recursive problems
  - Algorithms with "holes" given as higher order parameters
  - Algorithms that return other algorithms as higher order results
- This is extremely powerful, as it even allows us to reimplement apparently unrelated concepts such as SQL operators

Assignments in class and during the practicum

Assignments in class and during the practicum

Build and test, on paper and then in Python

- A `Car` class, with a `drive` function that returns the car at a new position
- A `driveAllCars` function that `drives` all cars in a list through the use of `map`
- A `removeArrived` function that removes all cars from the list that reached their destination through the use of `filter`

This is it!

The best of luck, and thanks for the attention!