Higher order functions

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

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


```
1 def combine(op,x,y):
2   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


```
1  def combine(op,x,y):
2    return op(x,y)
3
4  def plus(x,y): return x + y
5  def times(x,y): return x * y
6  def minus(x,y): return x - y
7
8  print(combine(plus, 10, 20))
9  print(combine(times, 10, 20))
10 print(combine(minus, 10, 20))
```

Example

- **What does this code do?**

HOF's


```
1  def combine(op,x,y):
2    return op(x,y)
3
4  def plus(x,y): return x + y
5  def times(x,y): return x * y
6  def minus(x,y): return x - y
7
8  print(combine(plus, 10, 20))
9  print(combine(times, 10, 20))
10 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


```
1  def combine(op,x,y):
2    return op(x,y)
3
4  def plus(x,y): return x + y
5  def times(x,y): return x * y
6  def minus(x,y): return x - y
7
8  print(combine(plus, "10", "20"))
9  print(combine(times, 10, 20))
10 print(combine(minus, 10, 20))
```

Example

- **What does this code do?**

HOF's


```
1  def combine(op,x,y):
2    return op(x,y)
3
4  def plus(x,y): return x + y
5  def times(x,y): return x * y
6  def minus(x,y): return x - y
7
8  print(combine(plus, "10", "20"))
9  print(combine(times, 10, 20))
10 print(combine(minus, 10, 20))
```

Example

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

HOF's


```
1  def combine(op,x,y):
2    return op(x,y)
3
4  def plus(x,y): return x + y
5  def times(x,y): return x * y
6  def minus(x,y): return x - y
7
8  print(combine(plus, "10", "20"))
9  print(combine(times, 10, 20))
10 print(combine(minus, 10, 20))
```

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

HOF's


```
1  def combine(op,x,y):
2    return op(x,y)
3
4  def plus(x,y): return x + y
5  def times(x,y): return x * y
6  def minus(x,y): return x - y
7
8  print(combine(plus, "10", "20"))
9  print(combine(times, 10, 20))
10 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


```
1  def combine(op,x,y):
2    return op(x,y)
3
4  print(combine((lambda x,y: x+y), "10", "20"))
5  print(combine((lambda x,y: x*y), 10, 20))
6  print(combine((lambda x,y: x-y), 10, 20))
```

Lambda-syntax function definition

- **What does this code do?**

HOF's


```
1  def combine(op,x,y):
2    return op(x,y)
3
4  print(combine((lambda x,y: x+y), "10", "20"))
5  print(combine((lambda x,y: x*y), 10, 20))
6  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


```
1  def combine(op,x,y):
2    return op(x,y)
3
4  print(combine((lambda x,y: x+y), "10", "20"))
5  print(combine((lambda x,y: x*y), 10, 20))
6  print(combine((lambda x,y: x-y), 10, 20))
```

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

HOF's

```
1  def combine(op,x,y):
2    return op(x,y)
3
4  print(combine((lambda x,y: x+y), "10", "20"))
5  print(combine((lambda x,y: x*y), 10, 20))
6  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


```
1  def combine(op,x,y):
2    return op(x,y)
3
4  def choose_operation():
5    i = input("Choose an operation between +, -, or *")
6    if i == "+":
7      return lambda x,y: x+y
8    elif i == "-":
9      return lambda x,y: x-y
10   else:
11     return lambda x,y: x*y
12 print(combine(choose_operation(), 10, 20))
```

Lambda-syntax function definition

- **What does this code do?**

HOF's


```
1  def combine(op,x,y):
2    return op(x,y)
3
4  def choose_operation():
5    i = input("Choose an operation between +, -, or *")
6    if i == "+":
7      return lambda x,y: x+y
8    elif i == "-":
9      return lambda x,y: x-y
10   else:
11     return lambda x,y: x*y
12 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


```
1  def combine(op,x,y):
2    return op(x,y)
3
4  def choose_operation():
5    i = input("Choose␣an␣operation␣between␣+,␣-,␣or␣*")
6    if i == "+":
7      return lambda x,y: x+y
8    elif i == "-":
9      return lambda x,y: x-y
10   else:
11     return lambda x,y: x*y
12 print(combine(choose_operation(), 10, 20))
```

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

HOF's

```
1  def combine(op,x,y):
2    return op(x,y)
3
4  def choose_operation():
5    i = input("Choose an operation between +, -, or *")
6    if i == "+":
7      return lambda x,y: x+y
8    elif i == "-":
9      return lambda x,y: x-y
10   else:
11     return lambda x,y: x*y
12 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

Assignments

Assignments

Build and test, on paper...

- A hof `map` that transforms all elements of a list:
  - `map(lambda x: x + 10, Node(1,Node(2,Node(3,Empty)))) -> Node(11,Node(12,Node(13,Empty)))`
- A hof `filter` that removes elements from a list:
  - `filter(lambda x: x % 2 == 0, Node(1,Node(2,Node(3,Empty)))) -> Node(2,Empty)`
- A hof `reduce` that condenses a list into a single value:
  - `reduce(lambda x,y: x + y, Node(1,Node(2,Node(3,Empty)))) -> 6`

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 it!

The best of luck, and thanks for the attention!